ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERÍA DEL SOFTWARE

Servicio multiplataforma para la consulta y gestión de contenido multimedia

Multiplatform service for consultation and manage multimedia content

Realizado por
Jesús Garrido Moscoso
Tutorizado por
Eduardo Guzmán De los Riscos
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, DICIEMBRE 2016

Fecha defensa:
El Secretario del Tribunal

Resumen: En la actualidad, el consumo de *streaming* de vídeo bajo demanda está casi al mismo nivel que el de la televisión convencional, solo 5 puntos por debajo de ésta. A esto añadimos la multitud de plataformas multimedia existentes para ver estos contenidos, provocando una gran fragmentación del material que visualizas, la pérdida de listas clasificadas o preferencias y la repetición de contenido entre una plataforma u otra. Este proyecto trata de solucionar estos problemas con el desarrollo de una aplicación web y móvil, para los sistemas operativos *Android* e *iOS*. Su objetivo es centralizar toda la información clasificada y visualizada por un usuario en nuestra plataforma, ofreciendo una experiencia sencilla y usable para la clasificación de este contenido en listas de películas asociadas al usuario, que podría categorizar como: vistas, pendientes, favoritas y *blacklist*. Además, la plataforma ofrecerá la búsqueda de películas por su título y la posibilidad de consultar la información detallada de una película, incluidas las puntuaciones o *ratings,* recogidas de las plataformas de puntuaciones más importantes y conocidas por los usuarios.

Palabras clave: móvil, aplicación, página, araña, *Django*, *API*, *REST*, *script*, *Python*, *Android*, *Material*, nativo, *Java*, *PostgreSQL*, película, multimedia, clasificación, puntuación, vista, pendiente, favorita.

Abstract: Nowadays, the consumption of streaming video on demand is almost at the same level that which concerned to conventional television, only 5 points below this. Also, we have to take into account the large number of existing multimedia platforms to watch those topics. It leads to a great fragmentation of the material that you can watch, the loss of the classified list and the repetition of the topic among a platform and another one. This project tries to fix those problems with the creation of a web and mobile environment for *Android* and *iOS*. Its aim is to focus all the classified and watched information by an user in our platform, offering an useful experience for the classification of this topic in film lists associated to the user, which could be: seen, watchlist, favorites and blacklist. Besides, the platform will offer the search of films considering the tittle and the possibility of consulting the detailed information of a film, including ratings, getting from the platforms with the most important ratings and known by the users.

Keywords: app, mobile, application, web, scrapper, Django, API, REST, script, Python, Android, Material, native, Java, PostgreSQL, movie, multimedia, film, rating, seen, watchlist, favorites, blacklist.

Índice general

Capítulo 1. Introducción

1.1 Motivación

En los últimos años han aparecido grandes novedades tecnológicas como son los *smartphones*, *tablets*, *Smart TVs*… Gracias a ello, muchos de los contenidos e información que consumimos se visualizan a través de estos dispositivos. Hasta ahora, un usuario tenía que esperar a una hora y día concretos de la semana para ver una serie o película. Actualmente, el consumo de contenidos digitales a la carta, está en aumento, a través de plataformas *streaming* como *Netflix*, *Wuaki*, *Yomvi*… Con estos nuevos servicios, ha cambiado totalmente la forma y demanda del contenido digital en Internet; ahora es el usuario el que decide qué contenido y cuándo lo quiere ver.

Según un informe de Ericsson, el consumo de *streaming* de vídeo bajo demanda está casi al mismo nivel que el de la televisión convencional: 80% frente a 85%, como podemos apreciar en la ***Figura 1.1***.

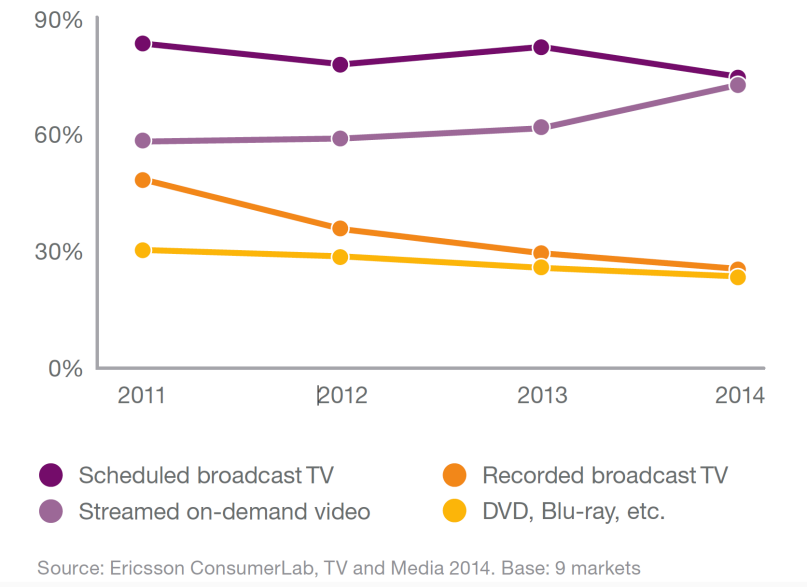
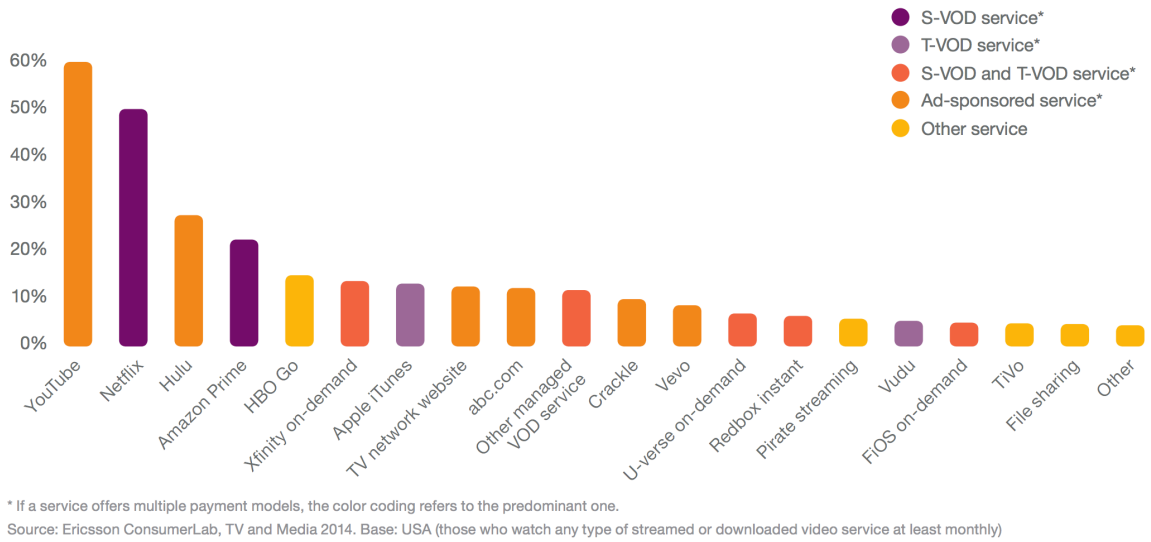


Figura 1.1 Porcentaje de personas que ven diferentes tipos de medios de comunicación más de una vez por semana

De los usuarios que consumen contenido bajo demanda, lo hacen a través de una gran variedad de plataformas con diferente catálogo, como podemos apreciar en la ***Figura 1.2***. Esto provoca una gran fragmentación del contenido que se visualiza, por lo que las listas de contenido visualizado o preferencias se perderán, provocando que haya contenido repetido y sin clasificar entre una plataforma u otra.



* If a service offers multiple payment models, the color coding refers to the predominant one.

Source: Ericsson ConsumerLab, TV and Media 2014. Base: USA (those who watch any type of streamed or downloaded video service at least monthly)

Figura 1.2 Porcentaje de consumidores que utilizan actualmente cada servicio bajo demanda en Estados Unidos

Como hemos visto antes, y debido al gran aumento del consumo de contenido bajo demanda, en estos tiempos, se plantea un problema (y a su vez el reto) de cara a mantener tu información centralizada en un mismo medio.

1.2 Objetivos

A raíz de estos problemas y la gran cantidad de información fragmentada en Internet, surgió la idea de desarrollar una aplicación multiplataforma, con el objetivo de aportar a los usuarios que su información esté centralizada, y una forma de clasificar su contenido de una manera rápida y sencilla. Permitiendo, además, el acceso a la información detallada de una película como su sinopsis, duración, fecha de estreno, reparto…

Por esta razón, se ha desarrollado una aplicación llamada *Mooviest*, pensada para que los usuarios puedan clasificar su contenido, buscar y consultar toda la información detallada de películas, así como sus listas de películas vistas, favoritas, pendientes o que no le interesan.

Para cumplir estos objetivos y funcionalidades se ha desarrollado una aplicación web donde el usuario gestionaría su perfil y consultaría sus listas más cómodamente, si usa un ordenador o *PC*. Además, una *app Android* e *iOS*, que facilitaría la clasificación de películas a través de gestos sobre su carátula o botones, entre otras funcionalidades. Para ello, hemos definido una serie de iconos representativos, que se usarán para la clasificación mediante botones y unos gestos habituales, usados en otras *apps* conocidas, de tal forma, que al usuario le resulte familiar la funcionalidad de la aplicación.

Para guardar toda la información de las películas, se ha usado una base de datos en *PostgreSQL*, gestionada por un servidor en *Django*. Esta información ha sido obtenida mediante *APIs* de terceros; para el idioma español, se ha usado la *API* de *Tviso* y para inglés la de *Trakt.tv*. Para la obtención de puntuaciones de las diferentes plataformas más importantes, se ha hecho uso de *scrappers* (software para extraer

contenido de sitios web) ya que no disponían de *API* de la que obtener dicha información. Para aportar al proyecto mayor autonomía, se han desarrollado unos scripts que actualizarán el contenido de la plataforma automáticamente y de forma periódica, los datos de las películas, así como, las puntuaciones, recogidas de múltiples plataformas como son *Tviso*, *IMDb*, *filmaffinity*…

1.3 Materiales y tecnologías usadas

- *Hardware*
    - Portátil *MacBook Pro* Retina
    - Teléfono móvil *Xiaomi Redmi Note 3 Pro* con *Android* 6.0.1 *Marshmallow*
    - Teléfono móvil *Samsung Galaxy A3* con *Android* 6.0.1 *Marshmallow*
    - Teléfono móvil *LG G3s* con *Android* 4.4.2 *KitKat*
- *Software*
    - *Microsoft Word 2016*
    - Navegador web *Google Chrome*
    - *Android Studio*
    - *Android SDK*
    - Gestor de paquetes *pip*
    - Cliente *PostgreSQL*, *PSequel*
    - Cliente *REST*, *Advanced REST client*
    - *Adobe Photoshop CS6*

1.4 Contenido de la memoria

Este trabajo de fin de grado se ha divido en varios capítulos tratando los temas más importantes de cada fase de desarrollo. Se tratan los siguientes:

- **Capítulo 1. Introducción:** En esta sección se ha descrito, a grandes rasgos, el proyecto a realizar indicando los motivos por los que surge la necesidad de la aplicación, cuáles son los objetivos del mismo, materiales utilizados para conseguirlo y este breve resumen del contenido de esta memoria.

- **Capítulo 2. Tecnologías y herramientas utilizadas:** En este capítulo introducimos al lector en las diferentes tecnologías utilizadas para llevar a cabo todo el desarrollo software con definiciones de cada una y nociones básicas.

- **Capítulo 3. Especificación de requisitos:** Tratamos los detalles sobre las funciones que debe cumplir la aplicación con un análisis de requisitos (funcionales y no funcionales) que marca el camino del desarrollo.

- **Capítulo 4. Análisis y diseño:** Entramos en la primera fase antes de comenzar el desarrollo comentando los casos de uso que surgen tras el análisis de

requisitos, además de una especificación de las diferentes arquitecturas y base de datos que se aplican.

- **Capítulo 5. Implementación e instalación:** Se expondrá de manera más descriptiva, el grueso del trabajo que en este caso es todo el desarrollo. Se comienza hablando de la estructura e implementación de la base de datos, la creación de la API, los scripts para la recogida de datos, la creación y configuración del proyecto tanto web como móvil, y a partir de aquí explicaremos en detalle cada sección de la aplicación, comentando primero los elementos visuales; tras esto explicaremos el funcionamiento de cada elemento.

- **Capítulo 6. Conclusiones y trabajo futuro:** Por último, comentaremos las conclusiones que se pueden extraer de toda la elaboración de un proyecto y se discutirán posibles trabajos futuros que se podrían realizar.

- **Bibliografía:** Material bibliográfico que se ha usado para el desarrollo de la aplicación y la realización de esta memoria.

1.5 División del trabajo

Este proyecto lo hemos realizado un grupo de tres miembros. Se ha dividido en una parte común, el servidor *Django* y todos sus componentes y una parte individual, una aplicación en diferentes plataformas: web, *Android* e *iOS*. Como podemos observar en la ***Tabla 1.1,*** indicamos el porcentaje de trabajo realizado por cada miembro del grupo en las principales partes del proyecto.

Tabla 1.1 División del trabajo y porcentaje realizado por cada miembro

			Jesús Garrido Moscoso	**José Antonio Palacios Ramírez**	**Antonio Romero Gómez**
Servidor Django	Creación e instalación		10%	80%	10%
	API	Movie	30%	40%	30%
		User	10%	80%	10%
	Scripts	Scrappers	45%	10%	45%
		Tviso	45%	10%	45%
		Trakt.tv	45%	10%	45%
		Script principal	40%	20%	40%
Aplicación Android			100%		
Aplicación IOS					100%
Aplicación Web				100%	

Capítulo 2. Tecnologías y herramientas utilizadas

2.1 Python 3

Python es un lenguaje de programación interpretado, cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Es un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Como comentábamos es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Es administrado por la *Python Software Foundation*. Posee una licencia de código abierto compatible con la Licencia pública general de *GNU*. Fue creado a finales de los ochenta siendo *Van Rossum* el principal autor de *Python*.

La versión 3 incluye una serie de cambios que hace que requiera reescribir el código de versiones anteriores.

2.2 *pip*

Es un sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en *Python*.

Una ventaja importante de *pip*, es la facilidad de uso en línea de comandos, el cual permite instalar paquetes de software de *Python* fácilmente con el siguiente comando:

```
pip install nombre-paquete
```

Los usuarios también pueden fácilmente desinstalar algún paquete:

```
pip uninstall nombre-paquete
```

Además, *pip* tiene una característica para instalar listas de paquetes a través de un "archivo" de requisitos. Esto permite recrear un grupo entero de paquetes en un entorno separado como por ejemplo en otro ordenador o entorno virtual. Se realizaría a través de este comando:

```
pip install -r requisitos.txt
```

2.3 Django

Django es un *framework* de desarrollo web de código abierto, escrito en *Python* y que respeta el patrón de diseño conocido como Modelo-Vista-Controlador. La meta de *Django* es la creación de sitios webs complejos de forma ágil y fácil, poniendo énfasis en la reutilización, conectividad y extensibilidad de componentes y su principio no te repitas (*DRY, Don´t Repeat Yourself*).

2.4 Django REST Framework

Django REST Framework es una librería, que nos permite construir un *API REST* sobre *Django* de forma sencilla. Ofreciendo una alta gama de método y funciones para el manejo, definición y control de nuestros recursos.

Incluye varios aspectos importantes en el diseño y creación de *APIs* tales como la autenticación.

Los conceptos más usados durante la creación de nuestra *API REST* son los siguientes:

- ***Serializers***: Permite que los datos complejos, como consultas e instancias a tipos de datos nativos de *Python*, puedan ser fácilmente transmitidos en otro tipo de datos como *JSON* o *XML* y viceversa.

- ***Viewsets***: Permite combinar la lógica de un conjunto de controladores relacionados en una sola clase. En el *viewset* se realiza la manipulación de las peticiones *HTTP*, además de crear los diferentes *endpoints* de la *API*.

2.5 PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos objeto-relacional, distribuido bajo licencia *BSD* y con su código fuente disponible libremente. Sus principales características son las siguientes:

- **Alta concurrencia:** Permite que mientras un proceso realiza una escritura en una tabla, otros accedan a la misma sin necesidad de bloqueos.

- Amplia variedad de tipos nativos.

- **Funciones:** Nos permite definir funciones personalizadas a través de varios lenguajes muy utilizados como son *pgSQL*, *Java*, *PHP*, *Python*…

2.6 API REST

REST, *REpresentational State Transfer*, es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar *HTTP*. Nos permite crear servicios y aplicaciones que pueden ser usadas por cualquier dispositivo o cliente que entienda *HTTP*. Por lo tanto, *REST*, es el tipo de arquitectura más natural y estándar para crear *APIs (Application Programming Interface),* para servicios orientados a Internet.

Al trabajar con *HTTP*, para su uso correcto debemos trabajar con los siguientes métodos:

- **GET:** Para consultar y leer recursos
- **POST:** Para crear recursos
- **PUT:** Para editar recursos
- **DELETE:** Para eliminar recursos.
- **PATCH:** Para editar partes concretas de un recurso.

Además, se deben usar los códigos de error de *HTTP* para representar el resultado de una llamada a la *API*. Algunos de ellos son los siguientes:

- **200 OK:** Petición realizada correctamente.
- **201 *Created*:** Recurso creado correctamente
- **400 *Bad Request*:** Petición fallida.
- **401 *Unauthorized*:** No autorizado, autenticación requerida.
- **404 *Not found*:** Recurso no encontrado.
- **500 *Internal Server Error*:** Error interno del servidor.

2.7 Android

Android es un sistema operativo basado en el núcleo *Linux*. Fue diseñado principalmente para dispositivos móviles con pantalla táctil. Inicialmente fue desarrollado por *Android Inc*., empresa que *Google* respaldó económicamente y más tarde, en 2005, la compró. *Android* fue presentado en 2007 junto la fundación del *Open Handset Alliance* (un consorcio de compañías de hardware, software y telecomunicaciones) para avanzar en los estándares abiertos de los dispositivos móviles.

La estructura del sistema operativo *Android* se compone de aplicaciones que se ejecutan en un *framework Java* de aplicaciones orientadas a objetos, sobre el núcleo de las bibliotecas de *Java* en una máquina virtual *Dalvik* con compilación en tiempo de ejecución, hasta la versión 5.0, luego cambió al entorno *Android Runtime* (*ART*).

Las aplicaciones se desarrollan habitualmente en el lenguaje *Java* con *Android Software Development Kit* (*Android SDK*), pero están disponibles otras herramientas de desarrollo. Todas las aplicaciones están comprimidas en formato *APK.*

Capítulo 3. Especificación de requisitos

La especificación de requisitos es una descripción completa del comportamiento del sistema que se va a desarrollar. En ella se describen los servicios que ha de ofrecer el sistema y las restricciones asociadas a su funcionamiento. Esta descripción completa del comportamiento del sistema se clasifica en dos grupos:

- **Requisitos funcionales:** Identifican los servicios que el sistema debe proporcionar, cómo debe reaccionar en respuesta a entradas externas y cómo debe comportarse en situaciones particulares.
- **Requisitos no funcionales:** Estos requisitos nos dicen las restricciones que afectan a los servicios y al correcto funcionamiento del sistema, tales como restricciones de tiempo, estándares, etc.

A continuación, se comentará cómo se llevó a cabo la recopilación de los datos de diferentes *APIs* para, en base a estos, realizar los requisitos.

3.1 Recopilación de datos

El primer paso para la ejecución de este proyecto, fue realizar un estudio y búsqueda en Internet, para encontrar los diferentes sitios webs y plataformas que ofrecían *APIs* gratuitas y así obtener toda la información relacionada de las películas existentes.

Para ello estudiábamos qué información ofrecían de una película, si tenían una base de datos completa, etc. Toda esta información era necesaria para saber qué *API* utilizar y cuál sería el modelado final del sistema. En todo este proceso, se enviaron emails a algunos sitios *webs* para confirmar si podíamos recopilar sus datos en nuestra base de datos, o preguntar por la extensión de su contenido.

Además de esto necesitábamos recopilar la puntuación o *rating* de las diferentes webs más importantes para las películas. En este proceso, vimos cómo podíamos obtenerlas o si no las proporcionaba alguna plataforma de manera gratuita.

Para la recogida de información de películas, finalmente optamos por obtenerla de la *API* de *Tviso*, que nos proporcionaba la información en el idioma español; para inglés, escogimos *Trakt.tv* y, por último, para las puntuaciones de películas, desarrollamos *scrappers* específicos para las páginas de *filmaffinity*, *IMDb*, *Metacritic* y *RottenTomatoes*.

3.2 Aplicación Mooviest (Android)

3.2.1 Requisitos funcionales

- **RF01 - Registrarse en el sistema:** La aplicación debe permitir el registro de un usuario introduciendo la información requerida por el sistema.
    - **RF01.1 - Introducir su nombre de usuario:** El usuario podrá introducir su nombre de usuario que servirá como identificador del sistema.
    - **RF01.2 - Introducir su email:** El usuario podrá introducir su email que también servirá como identificador del sistema.
    - **RF01.3 - Introducir su contraseña:** El usuario podrá introducir su contraseña.
    - **RF01.4 - Volver a introducir su contraseña:** El usuario podrá volver a introducir su contraseña.
    - **RF01.5 - Error:** Si ha ocurrido algún error, faltan campos obligatorios, no cumple con las validaciones de los campos obligatorios o no hay conexión a Internet, se mostrará un mensaje con dicha información.

- **RF02 - Iniciar sesión en el sistema:** La aplicación debe permitir iniciar sesión en el sistema.
    - **RF02.1 - Introducir su nombre de usuario o correo electrónico:** El usuario podrá introducir su nombre de usuario o email.
    - **RF02.2 - Introducir su contraseña:** El usuario podrá introducir su contraseña.
    - **RF02.3 - Error:** Si ha ocurrido algún error, no hay conexión a Internet, se mostrará un mensaje con dicha información.

- **RF03 - Cerrar sesión en el sistema:** La aplicación permitirá cerrar la sesión del usuario que la había iniciado o se había registrado, borrando así todos sus datos guardados en el dispositivo.

- **RF04 - Mostrar un menú lateral:** La aplicación permitirá abrir un menú lateral con todas las secciones de la aplicación, para poder cambiar de una a otra con facilidad.

- **RF05 - Mostrar el perfil del usuario:** La aplicación permitirá mostrar el perfil del usuario que ha iniciado sesión o se ha registrado.

- **RF06 - Mostrar ajustes del perfil del usuario:** La aplicación permitirá mostrar los ajustes del perfil del usuario.

- **RF07 - Modificar el perfil del usuario:** La aplicación permitirá modificar el perfil del usuario que ha iniciado sesión o se ha registrado.

- **RF07.1 - Modificar la imagen del perfil del usuario:** Podrá modificar la imagen del perfil del usuario, accediendo a una imagen guardada en su dispositivo.
- **RF07.2 - Modificar su nombre de usuario:** Podrá modificar su nombre de usuario por otro no existente en el sistema.
- **RF07.3 - Introducir o modificar su nombre:** Podrá introducir o modificar su nombre.
- **RF07.4 - Introducir o modificar sus apellidos:** Podrá introducir o modificar sus apellidos.
- **RF07.5 - Modificar su email:** Podrá modificar su email por otro no existente en el sistema.
- **RF07.6 - Introducir o modificar su fecha de nacimiento:** Podrá introducir o modificar su fecha de nacimiento, mediante un calendario.
- **RF07.7 - Introducir o modificar su ciudad:** Podrá introducir o modificar su ciudad.
- **RF07.8 - Introducir o modificar su código postal:** Podrá introducir o modificar su código postal.
- **RF07.9 - Error:** Si ha ocurrido un error, faltan campos obligatorios, ya existe otro usuario en el sistema con el mismo nombre de usuario o email, se mostrará un mensaje con dicha información.

- **RF08 - Mostrar un sistema de clasificación de películas:** La aplicación permitirá mostrar un sistema de clasificación de películas para el usuario que ya esté registrado o haya iniciado sesión, es decir se mostrará la carátula de una película y unos botones para su clasificación.
    - **RF08.1 - Clasificar una película realizando un gesto o *swipe* sobre ella:** La aplicación permitirá clasificar una película realizando unos determinados gestos.
        - **RF08.1.1 - Clasificar una película a la lista de películas favoritas del usuario:** La aplicación permitirá, mediante un gesto hacia arriba sobre la película, clasificarla en la lista de películas favoritas del usuario.
        - **RF08.1.2 - Clasificar una película a la lista de películas pendientes del usuario:** La aplicación permitirá, mediante un gesto hacia la izquierda sobre la película, clasificarla en la lista de películas pendientes del usuario.
        - **RF08.1.3 - Clasificar una película a la lista de películas vistas del usuario:** La aplicación permitirá, mediante un gesto hacia la derecha sobre la película, clasificarla en la lista de películas vistas del usuario.
        - **RF08.1.4 - Clasificar una película a la lista negra (*blacklist)* del usuario:** La aplicación permitirá, mediante un gesto hacia

abajo sobre la película, clasificarla en la *blacklist* de películas del usuario, es decir, en una lista de películas que no le interesan.

- **RF08.2 - Clasificar una película pulsando uno de los cuatro botones visibles:** La aplicación permitirá clasificar una película pulsando uno de los cuatro botones visibles para el usuario.
    - **RF08.1.1 - Clasificar una película a la lista de películas favoritas del usuario:** La aplicación permitirá, pulsando un botón con el icono de una estrella, clasificarla en la lista de películas favoritas del usuario.
    - **RF08.1.2 - Clasificar una película a la lista de películas pendientes del usuario:** La aplicación permitirá, pulsando un botón con el icono de un marcapáginas, clasificarla en la lista de películas pendientes del usuario.
    - **RF08.1.3 - Clasificar una película a la lista de películas vistas del usuario:** La aplicación permitirá, pulsando un botón con el icono de un ojo, clasificarla en la lista de películas vistas del usuario.
    - **RF08.1.4 - Clasificar una película a la lista negra (*blacklist)* del usuario:** La aplicación permitirá, pulsando un botón con el icono de una cruz, clasificarla en la *blacklist* de películas del usuario.

- **RF09 - Mostrar la previsualización de las listas de películas clasificadas por el usuario:** La aplicación permitirá mostrar una previsualización de 10 películas de cada lista del usuario.

- **RF10 - Mostrar las listas de películas clasificadas por el usuario:** La aplicación permitirá mostrar las listas de películas clasificadas del usuario que ha iniciado sesión o se ha registrado.
    - **RF10.1 - Mostrar la lista de películas vistas por el usuario:** La aplicación permitirá mostrar la lista de películas vistas por el usuario.
    - **RF10.2 - Mostrar la lista de películas pendientes del usuario:** La aplicación permitirá mostrar la lista de películas pendientes del usuario.
    - **RF10.3 - Mostrar la lista de películas favoritas del usuario:** La aplicación permitirá mostrar la lista de películas favoritas del usuario.
    - **RF10.4 - Mostrar la lista negra de películas (*blacklist*) del usuario:** La aplicación permitirá mostrar la *blacklist* de películas del usuario.

- **RF11 - Mostar la página de búsqueda de películas:** La aplicación permitirá mostrar la página de búsqueda de películas.

- **RF12 - Buscar películas por su título original o el título del idioma del usuario:** La aplicación permitirá buscar películas por su título original o el título del idioma del usuario.

- **RF12.1 - Introducir o modificar el campo de búsqueda de una película:** Podrá introducir o modificar el campo de búsqueda de una película.
- **RF12.2 - Borrar el campo de búsqueda de una película:** Podrá, a través de un botón con el icono de una cruz, borrar el contenido del campo de búsqueda de una película.
- **RF12.3 - Aviso:** Si no se ha encontrado ningún resultado o no hay conexión a internet, se mostrará un mensaje con dicha información.

- **RF13 - Mostrar la información detallada de una película:** La aplicación permitirá mostrar la información detallada de una película.
  - **RF13.1 - Puntuaciones o *ratings* de diferentes páginas web y plataformas:** Se mostrarán las puntuaciones de la película de diferentes páginas web y plataformas.
    - **RF13.1.1 - Mostrar la puntuación de la plataforma *Tviso*:** La aplicación permitirá mostrar la puntuación de la película de la plataforma *Tviso*.
    - **RF13.1.1 - Mostrar la puntuación de la plataforma *IMDb*:** La aplicación permitirá mostrar la puntuación de la película de la plataforma *IMDb*.
    - **RF13.1.1 - Mostrar la puntuación de nuestra plataforma:** La aplicación permitirá mostrar la puntuación de nuestra plataforma, que será la media de las demás puntuaciones.
  - **RF13.2 - Título de la película en el idioma del usuario:** Se mostrará el título de la película en el idioma del usuario.
  - **RF13.3 - Sinopsis:** Se mostrará la sinopsis de la película.
  - **RF13.4 - Duración:** Se mostrará la duración de la película.
  - **RF13.5 - Año:** Se mostrará el año de estreno: Se mostrará el año de estreno de la película.
  - **RF13.6 - Géneros:** Se mostrará un listado de géneros de la película.
  - **RF13.7 - Reparto:** Se mostrará un listado con el reparto de la película. La imagen del actor o director, su nombre real y el del papel en la película.
  - **RF13.8 - Clasificar la película mediante botones:** La aplicación permitirá clasificar la película mediante botones, al igual que el requisito RF08.2.

3.2.2 Requisitos no funcionales

- **RNF01 - Conexión a Internet:** La aplicación necesitará conexión a Internet para el acceso a la base de datos.

- **RNF02 - Adaptabilidad al dispositivo:** Debido a la gran cantidad de dispositivos móviles con diferentes pantallas que existen en el mercado, la interfaz de la aplicación debe ser adaptable (*responsive*), permitiendo así a cualquier usuario, usar las funciones y disfrutar de su experiencia sin importar el dispositivo que utilice.

- **RNF03 - Autenticación por token:** Para mayor seguridad y evitar que usuarios no autorizados en la aplicación accedan a información del contenido, hemos habilitado la autenticación por *token* para la realización de peticiones a la *API*.

- **RNF04 - Conexiones seguras *HTTPS*:** Para mayor seguridad en las transacciones entre el cliente y el servidor, se han habilitado conexiones seguras *HTTPS*, para intentar evitar la interceptación de información por parte de terceros en las conexiones.

- **RNF05 - Claves cifradas:** Para mayor seguridad en las cuentas de nuestros usuarios se han cifrado sus claves y así evitar el robo de cuentas.

- **RNF06 - Soporte multilenguaje:** La aplicación se mostrará en el idioma que el usuario tenga configurado su dispositivo, podrá ser inglés o español, en cualquier otro caso, se mostrará en inglés.

- **RNF07 - La base de datos se mantendrá actualizada con nuevo contenido periódicamente:** La aplicación permitirá la actualización e incorporación de contenido, a través de la ejecución automática y periódica de scripts.

Capítulo 4. Análisis y diseño

4.1 Casos de uso

En esta sección se muestran los diferentes casos de uso que derivan de los requisitos funcionales especificados en el ***Capítulo 3. Especificación de requisitos***.

4.1.1 Aplicación Mooviest (Android)

En la ***Figura 4.1***, podemos observar el diagrama de casos de uso principal de la aplicación Android. Para que no ocupara demasiado lo hemos dividido y en la ***Figura 4.2***, mostramos el caso de uso ***CU08*** ampliado.

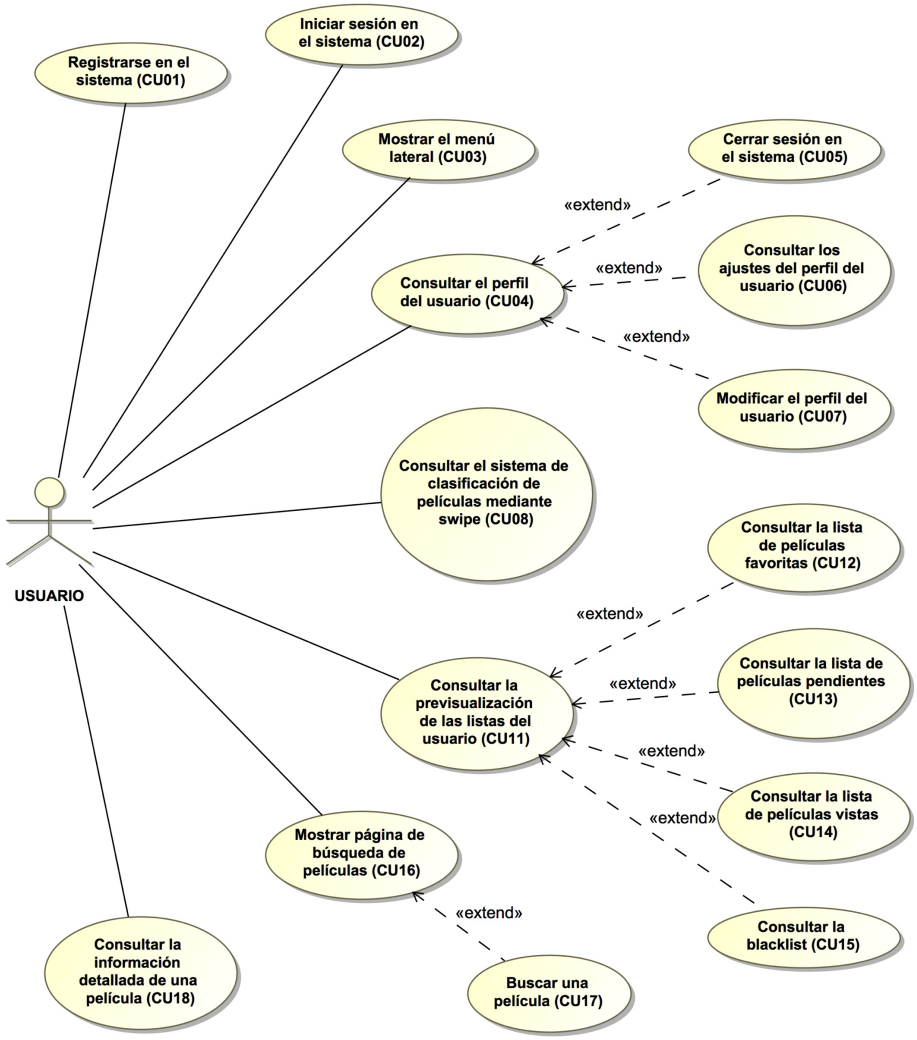


Figura 4.1 Diagrama de casos de uso principal de la aplicación Android

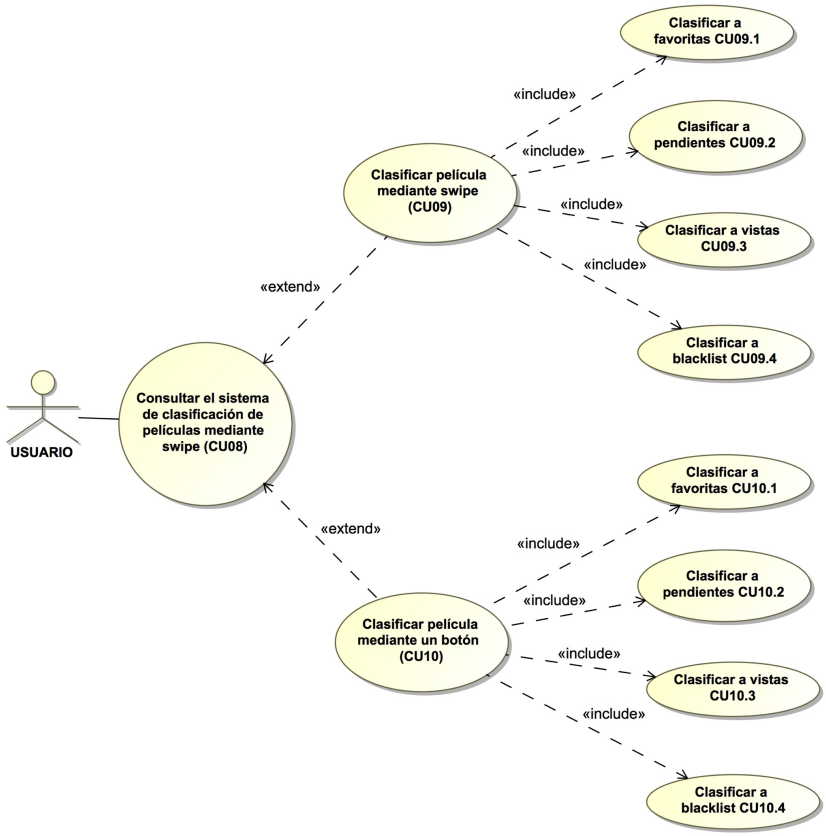


Figura 4.2 Ampliación del caso de uso CU08

CU01	**Registrarse en el sistema**	
Descripción	El usuario se registra en el sistema	
Precondición	El usuario ha abierto la aplicación	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa el enlace “¿No tiene cuenta todavía? Cree una”
	2	El sistema carga la página de registro del usuario
	3	El usuario introduce un nombre de usuario
	4	El usuario introduce su email
	5	El usuario introduce una contraseña
	6	El usuario vuelve a introducir su contraseña
	7	El usuario pulsa el botón “CREAR CUENTA”
	8	El sistema comprueba que los datos son válidos
	9	El sistema registra al usuario en el sistema
	10	El sistema obtiene los datos del usuario

	11	El sistema obtiene las películas del *swipe* y de las listas del usuario	
	12	El sistema carga la página principal de la aplicación (*Home*)	
Postcondición	El usuario se ha registrado en el sistema		
Escenario alternativo	**Paso**	**Acción**	
	8	Si los datos no son válidos	
		E1	El sistema informa que el nombre de usuario debe tener al menos tres caracteres
		E2	El sistema informa que el email no es válido
		E3	El sistema informa que la contraseña debe tener al menos seis caracteres alfanuméricos
		E4	El sistema informa que las contraseñas no coinciden

CU02	**Iniciar sesión en el sistema**		
Descripción	El usuario inicia sesión en el sistema		
Precondición	El usuario ha abierto la aplicación		
Escenario principal	**Paso**	**Acción**	
	1	El usuario introduce su email o nombre de usuario	
	2	El usuario introduce su contraseña	
	3	El usuario pulsa el botón “INICIAR SESIÓN”	
	4	El sistema comprueba que es un usuario existente	
	5	El sistema obtiene los datos del usuario	
	6	El sistema obtiene las películas del *swipe* y de las listas del usuario	
	7	El sistema carga la página principal de la aplicación (*Home*)	
Postcondición	El usuario ha iniciado sesión en el sistema		
Escenario alternativo	**Paso**	**Acción**	
	4	Si el usuario no existe en el sistema	
		E1	El sistema informa que el usuario no existe en el sistema

CU03	**Mostrar el menú lateral de la aplicación**	
Descripción	El usuario podrá ver el menú lateral de la aplicación	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el icono de la barra superior a la izquierda o desliza el dedo desde el extremo lateral izquierdo hacia la derecha de la pantalla.
	2	El sistema carga el menú lateral
Postcondición	El usuario ve el menú lateral	
Escenario alternativo	**Paso**	**Acción**

CU04	**Consultar el perfil del usuario**	
Descripción	El usuario podrá consultar su perfil	
Precondición	CU01 ó CU02 CU03	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre la sección del usuario
	2	El sistema carga la página del perfil del usuario
Postcondición	El usuario puede consultar su perfil	
Escenario alternativo	**Paso**	**Acción**

CU05	**Cerrar sesión en el sistema**	
Descripción	El usuario podrá cerrar sesión en el sistema	
Precondición	CU01 ó CU02 CU03 CU04	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el botón *power*

	2	El usuario pulsa sobre el enlace “Cerrar sesión”
	3	El sistema elimina las preferencias del usuario y la aplicación de la memoria del dispositivo
	4	El sistema carga la pantalla de inicio de sesión
Postcondición	El usuario ha cerrado sesión en el sistema	
Escenario alternativo	**Paso**	**Acción**

CU06	**Consultar los ajustes del perfil del usuario**	
Descripción	El usuario podrá consultar los ajustes de su perfil	
Precondición	CU01 ó CU02 CU03 CU04	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el botón editar (lápiz)
	2	El sistema carga la pantalla de ajustes del perfil del usuario
Postcondición	El usuario puede ver los ajustes de su perfil	
Escenario alternativo	**Paso**	**Acción**

CU07	**Modificar el perfil del usuario**	
Descripción	El usuario podrá modificar su perfil	
Precondición	CU01 ó CU02 CU03 CU04 CU06	
Escenario principal	**Paso**	**Acción**
	1	El usuario modifica su nombre de usuario
	2	El usuario introduce su nombre
	3	El usuario introduce sus apellidos
	4	El usuario modifica su email

	5	El usuario introduce su fecha de nacimiento	
	6	El usuario introduce su ciudad	
	7	El usuario introduce su código postal	
	8	El sistema comprueba que los datos son válidos	
	9	El sistema guarda los cambios en el sistema	
	10	El sistema carga la pantalla de consulta del perfil.	
Postcondición	El usuario ha modificado su perfil		
Escenario alternativo	**Paso**	**Acción**	
	8	Si los datos no son válidos	
		E1	El sistema informa que el nombre de usuario debe tener al menos tres caracteres
		E2	El sistema informa que el nombre de usuario no puede ser vacío
		E3	El sistema informa que el nombre de usuario ya existe en el sistema
		E4	El sistema informa que el email no es válido
		E5	El sistema informa que el email no puede ser vacío
		E6	El sistema informa que el email ya existe en el sistema

CU08	**Consultar el sistema de clasificación de películas (*swipe*)**	
Descripción	El usuario podrá consultar el sistema de clasificación de películas	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
Postcondición	El usuario consulta el sistema de clasificación de películas	
Escenario alternativo	**Paso**	**Acción**

CU09	**Clasificar una película mediante el sistema (*swipe*)**	
Descripción	El usuario podrá clasificar una película mediante el sistema de clasificación *swipe*	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
Postcondición	El usuario ha clasificado una película mediante el sistema de swipe	
Escenario alternativo	**Paso**	**Acción**

CU9.1	**Clasificar una película a la lista de películas favoritas con el sistema *swipe***	
Descripción	El usuario podrá clasificar una película a su lista de películas favoritas, arrastrándola hacia arriba en el sistema de clasificación	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
	1	El usuario desplaza la película hacia arriba
	2	El sistema clasifica la película en la lista de películas favoritas del usuario
	3	El sistema elimina la película de la vista
	4	El sistema carga otra película en la vista
Postcondición	El usuario ha clasificado una película en su lista de películas favoritas	
Escenario alternativo	**Paso**	**Acción**
	1	Si el usuario no ha desplazado la película lo suficiente hacia arriba
		E1 — El sistema devuelve la película a su posición inicial

CU9.2	**Clasificar una película a la lista de películas pendientes con el sistema *swipe***		
Descripción	El usuario podrá clasificar una película a su lista de películas pendientes, arrastrándola hacia la izquierda en el sistema de clasificación		
Precondición	CU01 ó CU02		
Escenario principal	**Paso**	**Acción**	
	1	El usuario desplaza la película hacia la izquierda	
	2	El sistema clasifica la película en la lista de películas pendientes del usuario	
	3	El sistema elimina la película de la vista	
	4	El sistema carga otra película en la vista	
Postcondición	El usuario ha clasificado una película en su lista de películas pendientes		
Escenario alternativo	**Paso**	**Acción**	
	1	Si el usuario no ha desplazado la película lo suficiente hacia la izquierda	
		E1	El sistema devuelve la película a su posición inicial

CU9.3	**Clasificar una película a la lista de películas vistas con el sistema *swipe***	
Descripción	El usuario podrá clasificar una película a su lista de películas vistas, arrastrándola hacia la derecha en el sistema de clasificación	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
	1	El usuario desplaza la película hacia la derecha
	2	El sistema clasifica la película en la lista de películas vistas del usuario
	3	El sistema elimina la película de la vista
	4	El sistema carga otra película en la vista

Postcondición	El usuario ha clasificado una película en su lista de películas vistas		
Escenario alternativo	**Paso**	**Acción**	
	1	Si el usuario no ha desplazado la película lo suficiente hacia la derecha	
		E1	El sistema devuelve la película a su posición inicial

CU9.4	**Clasificar una película a la lista de películas que no le interesan (*blacklist*) con el sistema *swipe***		
Descripción	El usuario podrá clasificar una película a su lista de películas que no le interesan, arrastrándola hacia abajo en el sistema de clasificación		
Precondición	CU01 ó CU02		
Escenario principal	**Paso**	**Acción**	
	1	El usuario desplaza la película hacia abajo	
	2	El sistema clasifica la película en su *blacklist*	
	3	El sistema elimina la película de la vista	
	4	El sistema carga otra película en la vista	
Postcondición	El usuario ha clasificado una película en su *blacklist*		
Escenario alternativo	**Paso**	**Acción**	
	1	Si el usuario no ha desplazado la película lo suficiente hacia abajo	
		E1	El sistema devuelve la película a su posición inicial

CU10	**Clasificar una película mediante el uso de botones**		
Descripción	El usuario podrá clasificar una película mediante el uso de botones		
Precondición	CU01 ó CU02		
Escenario principal	**Paso**	**Acción**	
Postcondición	El usuario ha clasificado una película mediante el uso de botones		

Escenario alternativo	Paso	Acción

CU10.1	Clasificar una película a la lista de películas favoritas con un botón	
Descripción	El usuario podrá clasificar una película a su lista de películas favoritas, pulsando un botón	
Precondición	CU01 ó CU02	
Escenario principal	Paso	Acción
	1	El usuario pulsa el botón con el icono de la estrella
	2	El sistema clasifica la película en la lista de películas favoritas del usuario
	3	El sistema elimina la película de la vista
	4	El sistema carga otra película en la vista
Postcondición	El usuario ha clasificado una película en su lista de películas favoritas	
Escenario alternativo	Paso	Acción

CU10.2	Clasificar una película a la lista de películas pendientes con un botón	
Descripción	El usuario podrá clasificar una película a su lista de películas pendientes, pulsando un botón	
Precondición	CU01 ó CU02	
Escenario principal	Paso	Acción
	1	El usuario pulsa el botón con el icono de un marcapáginas
	2	El sistema clasifica la película en la lista de películas pendientes del usuario
	3	El sistema elimina la película de la vista
	4	El sistema carga otra película en la vista
Postcondición	El usuario ha clasificado una película en su lista de películas pendientes	

Escenario alternativo	**Paso**	**Acción**

CU10.3	**Clasificar una película a la lista de películas vistas con un botón**	
Descripción	El usuario podrá clasificar una película a su lista de películas vistas, pulsando un botón	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa el botón con el icono de un ojo
	2	El sistema clasifica la película en la lista de películas vistas del usuario
	3	El sistema elimina la película de la vista
	4	El sistema carga otra película en la vista
Postcondición	El usuario ha clasificado una película en su lista de películas vistas	
Escenario alternativo	**Paso**	**Acción**

CU10.4	**Clasificar una película a la lista de películas que no le interesan (*blacklist*) con un botón**	
Descripción	El usuario podrá clasificar una película a su *blacklist*, pulsando un botón	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa el botón con el icono de una cruz
	2	El sistema clasifica la película en la *blacklist* del usuario
	3	El sistema elimina la película de la vista
	4	El sistema carga otra película en la vista
Postcondición	El usuario ha clasificado una película en su *blacklist*	
Escenario alternativo	**Paso**	**Acción**

CU11	**Consultar la previsualización de las listas de películas del usuario**	
Descripción	El usuario podrá consultar la previsualización de sus listas de películas	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el botón de las listas que se encuentra en la barra de *tabs*
	2	El sistema carga la página de previsualización de las listas del usuario
Postcondición	El usuario ve la previsualización de sus listas de películas	
Escenario alternativo	**Paso**	**Acción**

CU12	**Consultar la lista de películas favoritas del usuario**	
Descripción	El usuario podrá consultar su lista completa de películas favoritas	
Precondición	CU01 ó CU02 CU11	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el botón “MÁS” de la lista de películas favoritas
	2	El sistema carga la página con la lista de películas favoritas del usuario
Postcondición	El usuario ve su lista de películas favoritas	
Escenario alternativo	**Paso**	**Acción**

CU13	**Consultar la lista de películas pendientes del usuario**
Descripción	El usuario podrá consultar su lista completa de películas pendientes
Precondición	CU01 ó CU02 CU11

Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el botón “MÁS” de la lista de películas pendientes
	2	El sistema carga la página con la lista de películas pendientes del usuario
Postcondición	El usuario ve su lista de películas pendientes	
Escenario alternativo	**Paso**	**Acción**

CU14	**Consultar la lista de películas vistas del usuario**	
Descripción	El usuario podrá consultar su lista completa de películas vistas	
Precondición	CU01 ó CU02 CU11	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el botón “MÁS” de la lista de películas vistas
	2	El sistema carga la página con la lista de películas vistas del usuario
Postcondición	El usuario ve su lista de películas vistas	
Escenario alternativo	**Paso**	**Acción**

CU15	**Consultar la lista de películas que no le interesan (*blacklist*) del usuario**	
Descripción	El usuario podrá consultar su *blacklist* completa	
Precondición	CU01 ó CU02 CU11	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre el botón “MÁS” de la *blacklist*
	2	El sistema carga la página con la *blacklist* del usuario
Postcondición	El usuario ve su *blacklist*	

Escenario alternativo	Paso	Acción

CU16	Mostrar página de búsqueda de películas	
Descripción	El usuario podrá acceder a la página de búsqueda de películas	
Precondición	CU01 ó CU02	
Escenario principal	Paso	Acción
	1	El usuario pulsa sobre el botón con una lupa en la barra superior
	2	El sistema carga la página de búsqueda de películas
Postcondición	El usuario ve la página de búsqueda de películas	
Escenario alternativo	Paso	Acción

CU17	Buscar películas por su título		
Descripción	El usuario podrá buscar películas por su título		
Precondición	CU01 ó CU02		
Escenario principal	Paso	Acción	
	1	El usuario introduce un título	
	2	El sistema busca películas que contengan en su título esa palabra o palabras	
	3	El sistema muestra la lista de películas encontradas	
Postcondición	El usuario ve el resultado de las películas encontradas		
Escenario alternativo	Paso	Acción	
	2	Si el sistema no ha encontrado resultados	
		E1	El sistema informa que no ha encontrado resultados

CU18	Consultar la información detallada de una película	
Descripción	El usuario podrá consultar la información detallada de una película	
Precondición	CU01 ó CU02	
Escenario principal	**Paso**	**Acción**
	1	El usuario pulsa sobre una película del sistema de clasificación
	2	El sistema carga la página con la información detallada de la película.
Postcondición	El usuario ve la página de información detallada de una película	
Escenario alternativo	**Paso**	**Acción**

4.2 Arquitectura

Podríamos distinguir dos tipos de arquitecturas en este proyecto: por un lado, la arquitectura a alto nivel cliente-servidor entre los diferentes subsistemas, en la que los clientes serían las aplicaciones móviles y web y el servidor, en este caso lo hemos desarrollado en el *framework Django*. Por otro lado, en el caso de la aplicación *Android*, en cuanto al diseño de la aplicación y la organización de clases y funciones, se ha seguido el patrón Modelo Vista Controlador (MVC).

4.2.1 Arquitectura cliente-servidor.

La arquitectura cliente-servidor es un modelo de aplicación distribuida, en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes. Esta conexión se realiza cuando un cliente solicita información del servidor; en este momento, el cliente realiza una petición al servidor, y este le devuelve una respuesta.

En nuestro caso, nuestro servidor *Django* a través de *Django REST Framework*, nos suministrará un servicio: devolvernos la información solicitada por las apps de nuestra base de datos, para que cuando todos los clientes, que se encuentren en ejecución así lo soliciten, siempre tengamos a nuestro servidor respondiendo a todas las peticiones. En la ***Figura 4.3***, podemos ver el diagrama de la arquitectura cliente-servidor para nuestro sistema.

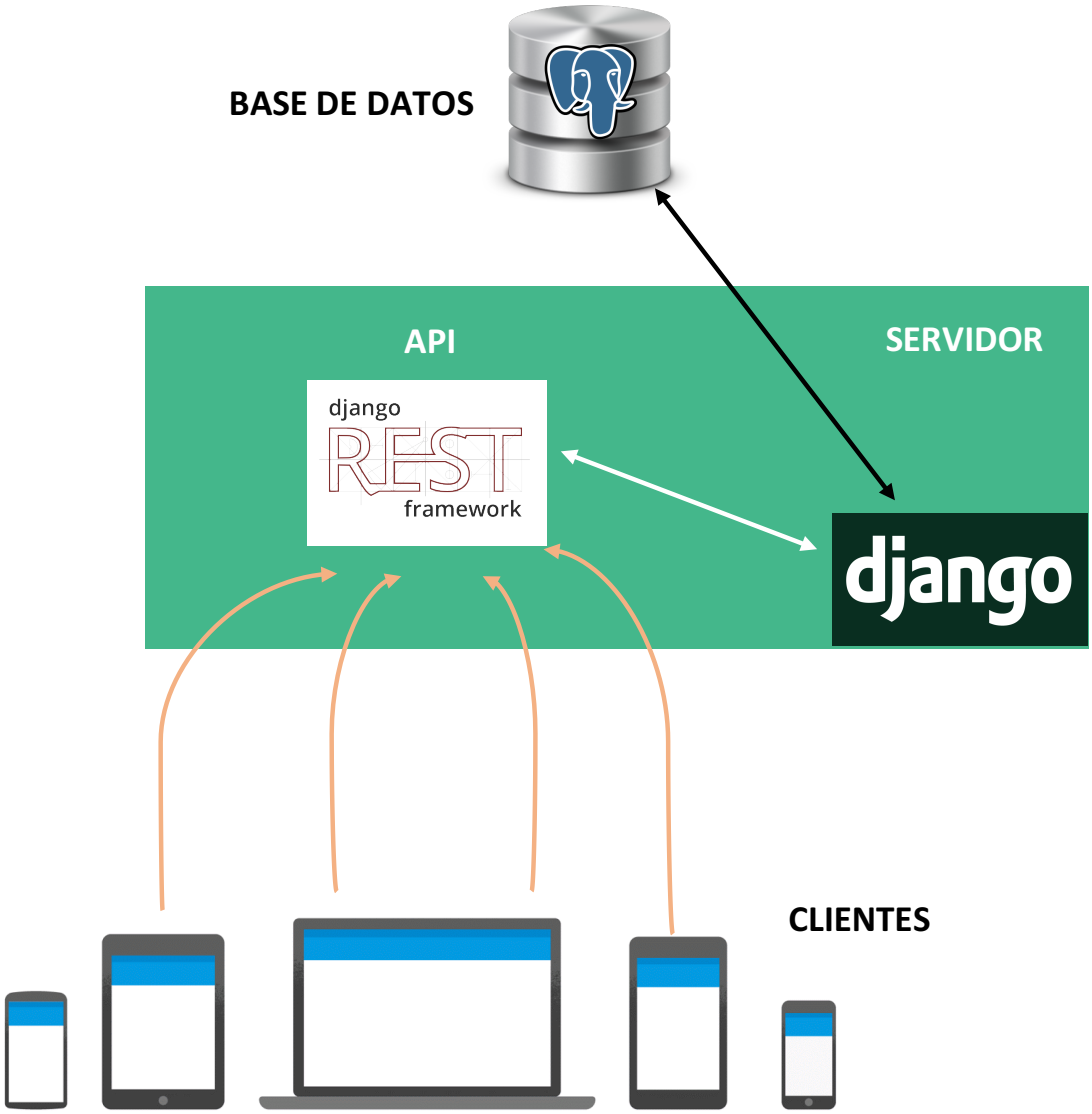


Figura 4.3 Diagrama de la arquitectura cliente-servidor de nuestro sistema

4.2.2 Arquitectura Modelo Vista Controlador

En nuestra app Android, seguimos el conocido patrón MVC. Un modelo muy usado en aplicaciones, donde se definen tres capas interconectadas para separar: el funcionamiento o lógica de la aplicación (Controlador), el tratamiento de los datos (Modelo), y la interfaz gráfica (Vista). En la ***Figura 4.4***, podemos observar el diagrama de la arquitectura MVC para nuestra aplicación.

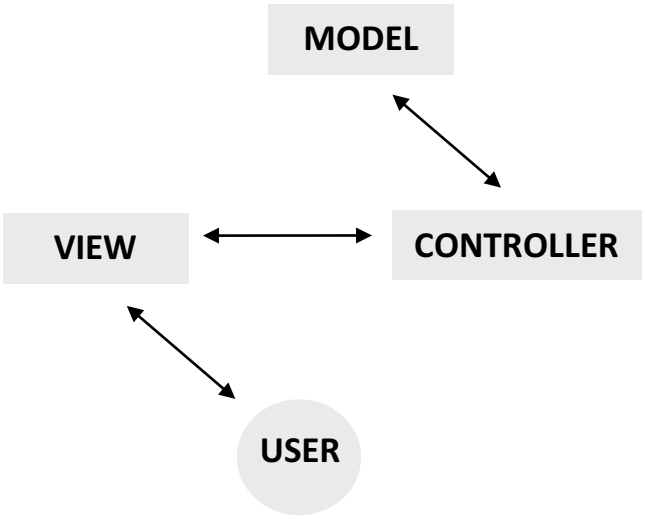


Figura 4.4 Diagrama de la arquitectura MVC de la aplicación Android

La forma en la que debe trabajar este patrón es conectando la *Vista*, que es la parte visible para el usuario de la aplicación, con un *Controlador*, que recibirá los datos de la *Vista* para tratarlos, o se los mandará a la *Vista* para mostrarlos al usuario. A su vez, este *Controlador* será el que se conecte con la capa del *Modelo*, para poder guardar los datos recibidos desde la *Vista*, o para solicitar los datos que queremos enviar a la *Vista*.

4.2.2.1 Modelo

Es la capa donde se trabaja con los datos, es decir, contiene los métodos para acceder a la información, en este caso, realizando peticiones a nuestra *API,* y las clases en *Java* para poder hacer uso de esta información en nuestra aplicación. Estos datos son los que tenemos almacenados en la base de datos *PostgreSQL* de nuestro servidor *Django*. Métodos como *getUserProfile()* o *updateUserProfile(),* permiten acceder a la librería *Retrofit*, que desempeña el rol de modelo dentro de nuestra arquitectura, para realizar las llamadas a la *API*.

Para poder hacer uso de la información obtenida de las respuestas de la *API* en nuestra aplicación, se han creado las clases necesarias en *Java,* mediante *POJO's*, basándose en el formato de las respuestas del *JSON*. Gracias a la librería *Gson,* que se integra en *Retrofit,* nos permite serializar y deserializar los objetos *JSON* en clases *Java* y viceversa.

4.2.2.2 Vista

La vista en *Android* se corresponde con archivos *XML*, que están asociados con *Activities* que las gestionan, pertenecientes al controlador. Las *activities,* son las encargadas de proporcionarles a la vista los datos que se mostrarán.

4.2.2.3 Controlador

En Android, existe un archivo llamado *AndroidManifest.xml*, donde se definen las *Activities*, mencionadas anteriormente*,* dentro de una clase *Application;* esta clase es la encargada de gestionar toda la aplicación y controlar las *activities*. De este modo todo el tráfico entre las vistas y el modelo, pasarán a través de esta clase *Application* controladora.

En las *activities*, se define la lógica de la aplicación, como hemos dicho antes, son las encargadas de proporcionarles a la vista los datos, además desde ellas, se llaman a otras clases que utilizamos para realizar las llamadas a la *API,* gestionar funcionalidades de las *activities*, etc.

4.3 Base de datos

Debido a la gran cantidad de información y relaciones posibles entre las tablas del modelo, se ha optado por utilizar una base de datos relacional, concretamente en PostgreSQL, que encaja perfectamente con Django, la tecnología utilizada en el lado del servidor.

En este proyecto necesitábamos relaciones entre un usuario y sus colecciones de películas, una película y sus *ratings,* las participaciones de los actores en esa película, etc.

Además, como podemos apreciar en la ***Figura 4.5***, cada tabla sensible de traducción, está relacionada con ***Lang***, para poder realizar el soporte multilenguaje. Es por ello que se complica un poco el modelado y se ha utilizado una base de datos relacional para abordar estas necesidades.

La tabla principal de nuestro modelo es ***Movie***, que está relacionada con todas las demás, como por ejemplo ***Saga***, ***Genre***, ***Movie_lang***, ***Rating***… Podríamos destacar la relación de ***Movie*** con ***Celebrity*** con una relación muchos a muchos, creando así una tabla intermedia llamada ***Participation***, encargada de proporcionar información sobre los premios de una *celebrity* en una película, su papel o su rol como actor, director, escritor en dicha película.

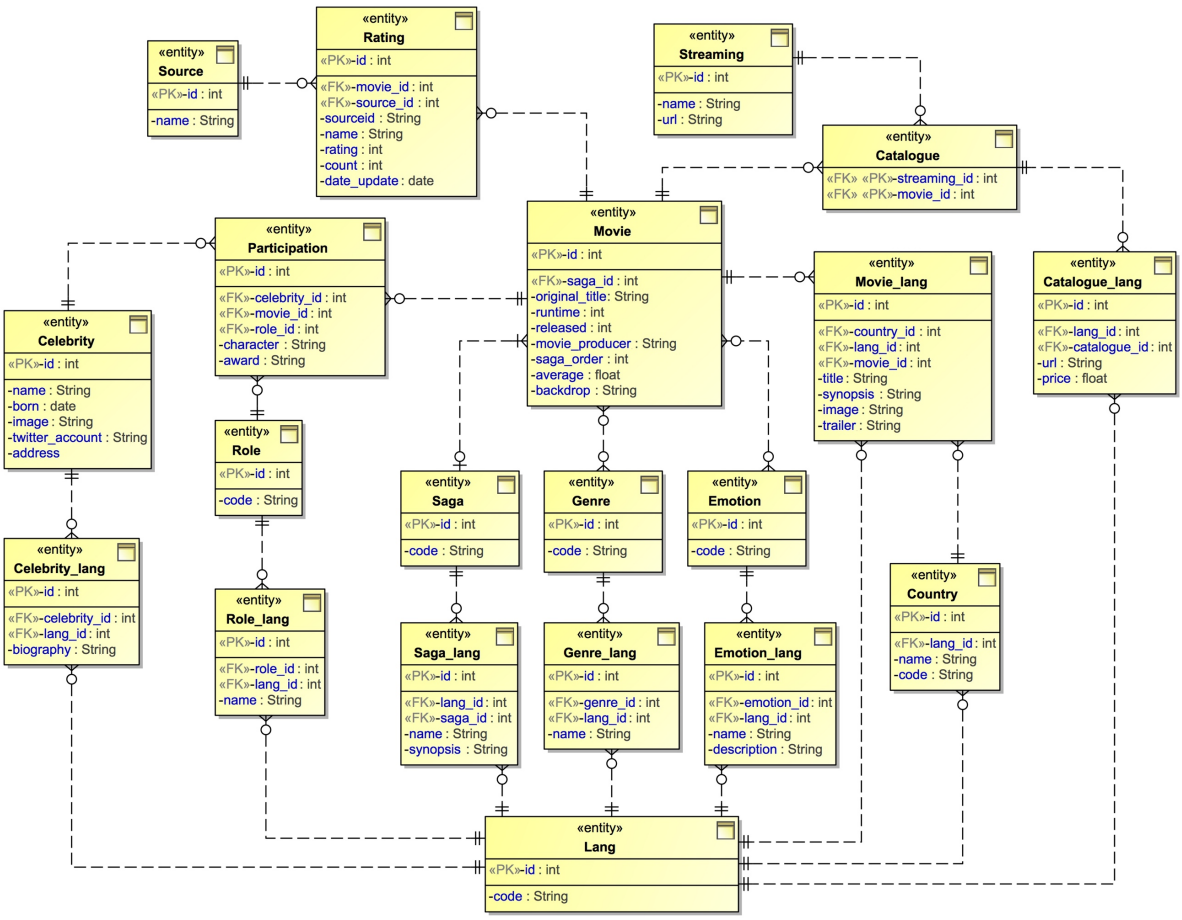


Figura 4.5 Diagrama E/R principal de la base de datos

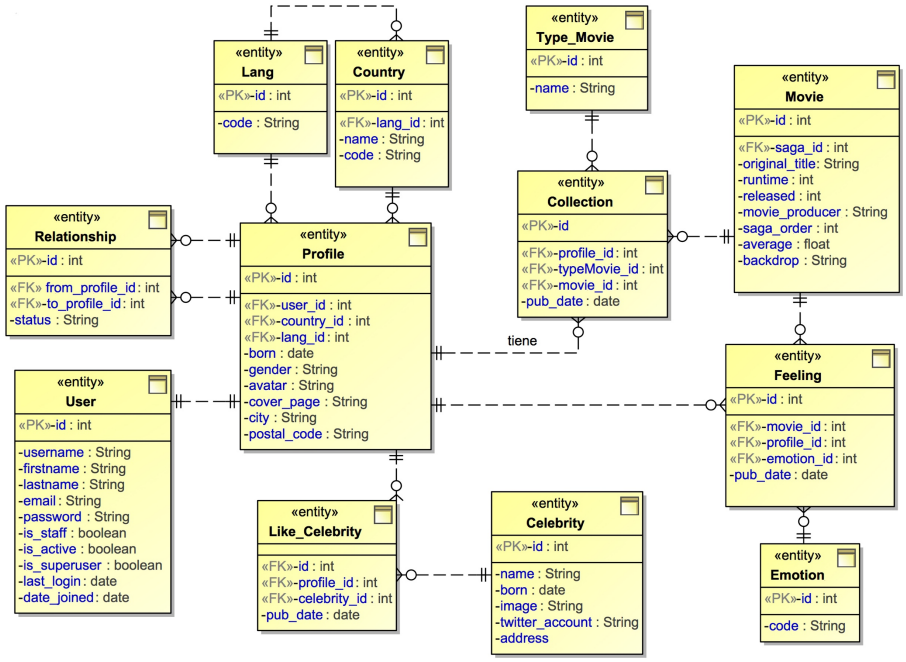


Figura 4.6 Diagrama E/R del usuario relacionado con el resto del modelo de la base de datos

En el esquema de la ***Figura 4.6,*** mostramos las relaciones del usuario con las demás tablas. Hemos separado la tabla ***User*** de ***Profile*** mediante una relación uno a uno. En ***User*** tenemos todos los atributos relacionados con el registro mientras que en ***Profile*** tenemos los atributos relacionados con la persona. De esta forma, es ***Profile*** la que se relaciona con las demás tablas.

Profile se relaciona consigo mismo mediante una relación muchos a muchos, que permitirá relacionar a usuarios de forma asimétrica, de forma que, un usuario podrá seguir a otro, pero éste otro no tiene por qué seguir al primero. También puede seguir a *celebrities* mediante la relación muchos a muchos con la tabla ***Celebrity***.

Además, ***Profile*** está relacionado con ***Lang***, para así poder mostrar toda la información en su idioma. Tiene asociadas colecciones para poder clasificar sus películas mediante una relación muchos a muchos con ***Movie***.

4.4 Estructura y diseño

Debido al uso de *Android*, para la interfaz de usuario y los estilos, se ha usado la guía de diseño de *Material Design*, que nos permite realizar una aplicación lo más familiar posible para al usuario y ser distinguida como una *app* nativa de *Android*.

El hecho de utilizar estilos nativos para la interfaz, es muy importante para el usuario final que utilizará la aplicación, ya que cualquiera, podrá adaptarse a la aplicación de manera rápida y comprender todo su significado, porque de esta forma sigue la misma estructura que las demás aplicaciones actuales.

En la ***Figura 4.7,*** podemos ver un ejemplo del uso de la guía de estilos Material Design, en este caso, para el desarrollo del menú lateral de la aplicación.

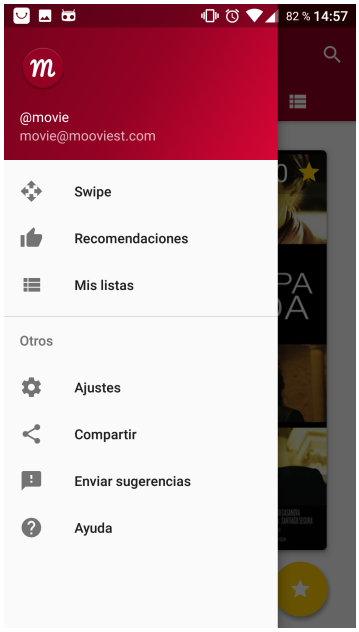


Figura 4.7 Menú lateral de la aplicación Android

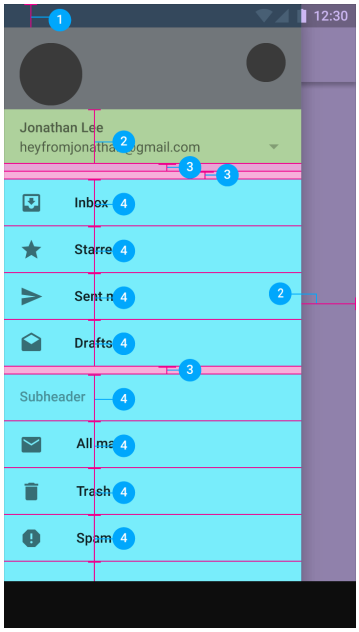


Figura 4.8 Guía de estilos Material Design para el desarrollo de un menú lateral

En cuando a los colores, se ha mantenido el *branding* de la marca *Mooviest*, donde creamos inicialmente su logo (***Figura 4.9***) y los colores personalizados basados en este.

Figura 4.9 Logo de la aplicación Android

Establecimos como colores primarios el blanco y rojo, que se pueden usar en la aplicación *Android* a través del fichero */res/values/colors.xml*, donde se guardan todo el conjunto de colores de una aplicación.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#940123</color>
    <color name="colorPrimaryButton">#80001d</color>
    <color name="colorPrimaryTrans">#ecd7dc</color>
    <color name="colorPrimaryDark">#4d0000</color>
    <color name="colorAccent">#FFFFFF</color>
    <color name="textColorPrimary">#FFFFFF</color>

                .
                .
                .

    <color name="seen">#3cca81</color>
    <color name="watchlist">#0090ff</color>
    <color name="favourite">#ffcc00</color>
    <color name="blacklist">#e74034</color>
</resources>
```

Capítulo 5. Implementación e instalación

5.1 API (Application Programming Interface)

Se ha desarrollado una *API*, cuyo objetivo es ofrecer a los clientes, que en nuestro caso serían la aplicación web y las móviles, la información alojada en nuestra base de datos *PostgreSQL* y que es gestionada por nuestro servidor *Django*.

5.1.1 Desarrollo

En este apartado explicaremos los conocimientos básicos para el desarrollo de una *API* usando *Django REST Framework.*

5.1.1.1 Autenticación

Comenzamos explicando el sistema de autenticación de nuestra aplicación, basado en ***token***. Cuando creamos un usuario en el sistema, se le asigna un *token* o identificador único, que será necesario para realizar las peticiones a nuestra *API*. Cualquier agente externo que no tenga un *token* del sistema asociado no podrá acceder a nuestro contenido.

```
from rest_framework.authtoken.models import Token
from rest_framework.authentication import TokenAuthentication
                .
                .
                .
# Método para registrar un usuario o iniciar sesión en el sistema
token = Token.objects.get_or_create(user=user)[0].key
                .
                .
                .
                .
```

5.1.1.2 Permisos

Los permisos determinan qué peticiones deben ser aceptadas o denegadas de un usuario. Se declaran en los ***Viewsets***, que son los encargados de recibir las peticiones de la *API*.

En *Django REST Framework*, hay muchos tipos de permisos por defecto, como son los siguientes entre otros:

- ***AllowAny:*** No restringe el acceso a ningún usuario.
- ***IsAuthenticated:*** Deniega el acceso a todos aquellos usuarios que no estén autenticados, en concreto para nuestro proyecto, por medio del sistema *token* citado anteriormente.

- ***IsAdminUser:*** Restringe el acceso a todos aquellos usuarios que no tengan el rol de administrador.

Para utilizar tanto los permisos como nuestro sistema de autenticación basado en token, los declararíamos en la clase del *viewset* correspondiente de la siguiente forma.

```
# Viewset para usuarios
class UserViewSet(GenericViewSet):

    authentication_classes = (TokenAuthentication,)
    permission_classes = (UserPermission,)
                .
                .
                .
```

5.1.1.3 Serializers

Los serializadores o ***serializers***, permiten convertir datos complejos como instancias del modelo de *Python* o consultas a *JSON*, *XML* u otros tipos de datos. Los serializers de *Django REST Framework*, tienen muchos campos predefinidos que nos permiten validar automáticamente ciertos tipos de datos como, por ejemplo, un correo electrónico, un número de caracteres o si un campo es requerido o no.

```
class UserSerializer(serializers.ModelSerializer):
    profile = ProfileSerializer(partial=True)

    class Meta:
        model = User
        fields = ('id', 'username', 'first_name', 'last_name',
'email', 'profile')

    def update(self, instance, validated_data):
        profile_data = validated_data.pop('profile')

        profile = instance.profile

        instance.username = validated_data.get('username',
instance.username)
        instance.first_name = validated_data.get('first_name',
instance.first_name)
        instance.last_name = validated_data.get('last_name',
instance.last_name)
        instance.email = validated_data.get('email',
instance.email)

        instance.save()

        profile.born = profile_data.get('born', profile.born)
```

```
        profile.gender = profile_data.get('gender',
profile.gender)
        profile.avatar = profile_data.get('avatar',
profile.avatar)
        profile.city = profile_data.get('city', profile.city)
        profile.postalCode = profile_data.get('postalCode',
profile.postalCode)

        profile.save()

        return instance
```

Como podemos observar en los métodos *validated_data.get(),* los campos de un usuario al actualizarlo, los valida Django de su modelo pudiendo ser ***CharField***, ***IntegerField*** o ***EmailField***, como hemos comentado anteriormente para comprobar si un correo es válido. Además, los *serializers* también nos permiten:

- crear y guardar instancias con los métodos *create()* y *update()*,
- realizar validaciones personalizadas de un campo concreto, como veremos a continuación, para comprobar que no existe otro usuario en el sistema con un *username* o nombre de usuario igual al introducido.

```
   def validate_username(self, data):
        user = User.objects.filter(username=data.lower())
        if self.instance is not None and self.instance.username !=
data and User.objects.filter(username=data.lower()):
            raise serializers.ValidationError('Nombre de usuario
ya registrado.')
        else:
            return data
```

5.1.1.4 Viewsets

Los ***viewsets***, gestionan las peticiones de la API y, para ello, tienen manejadores como *list(), create(), retrieve(), update(), partial_update()* o *destroy()*, que nos permiten identificarlos con los métodos HTTP conocidos (GET, POST, PUT…). También son los encargados de devolver una respuesta a la petición en el formato deseado, en nuestro caso *JSON*.

```
class UserViewSet(GenericViewSet):

    authentication_classes = (TokenAuthentication,)
    permission_classes = (UserPermission,)

                .
                .
```

```
                .

    def retrieve(self,request,pk):
        user = User.objects.get(pk=pk)
        profile = user.profile
        lang = Lang.objects.get(pk = profile.lang.id)

        return Response(
            {
                'user':{
                    'id': user.id,
                    'username': user.username,
                    'first_name': user.first_name,
                    'last_name': user.last_name,
                    'email': user.email,
                    'profile':{
                        'born': profile.born,
                        'avatar': profile.avatar.url,
                        'city': profile.city,
                        'gender': profile.gender,
                        'postalCode': profile.postalCode,
                        'lang': {
                            'code': lang.code
                        },
                    },
                },
                'status':status.HTTP_200_OK,
            }
        )

    def update(self,request,pk=None):
        data = request.data
        http_code = ""
        errors = None

        user = get_object_or_404(User,pk=pk)
        serializer = UserSerializer(instance=user,data=data)

        if serializer.is_valid():
            user = serializer.save()
            data =  serializer.data
            http_code = status.HTTP_200_OK

        else:
            data = None
            http_code = status.HTTP_400_BAD_REQUEST
            errors = serializer.errors

        return Response(
```

```
            {
                'user': data,
                'status': http_code,
                'errors': errors
            }
        )
```

En el método *retrieve*, equivalente a un *GET* del usuario, realizamos una consulta para obtener un usuario por su *id*, y lo devolvemos en formato *JSON*.

En el método *update*, equivalente a un *PUT* o *PATCH*, obtenemos también el usuario por su *id*, llamamos al *serializer* del usuario *UserSerializer* el cual nos validará los campos y, si son correctos, guardará esos datos y nos devolverá la respuesta del resultado en formato *JSON*.

5.1.2 Peticiones

Explicaremos las llamadas a la *API* de nuestro proyecto que hemos necesitado para que las aplicaciones se puedan comunicar con nuestro servidor y, en concreto, con la base de datos. Primero detallaremos las peticiones relacionadas con un usuario, para posteriormente pasar a las relacionadas con una película.

5.1.2.1 User

- **AUTH - PostLogin:** Iniciar sesión en el sistema.

  `(POST) /users/login/`

  Datos del formulario para x-www-form-urlencoded

Campo	Tipo	Descripción
username	*Text*	Nombre de usuario o email registrado en el sistema
password	*Text*	Contraseña asociada al usuario o email introducido

  Respuesta

```
{
   "message":"Login successfully",
   "status":200,
   "user":{
      "username":"movie",
```

```
        "profile":{
           "avatar":"/media/user/default/no-image.png",
           "lang":{
              "code":"es"
           }
        },
        "id":9,
        "email":"movie@mooviest.com"
     },
     "token":"8a0ac632536d7d1ac5db90f6f05338cef2778516"
}
```


Obtenemos los datos del usuario y su perfil, además del *token*, necesario para realizar el resto de peticiones que no son de tipo *AUTH* (Autorizadas).

Error 4xx


```
{
  "message": "User or password incorrect",
  "status": 404,
  "user": null,
  "token": null
}
```


Si no existe un usuario en el sistema con los campos introducidos nos devolverá un mensaje de error.

- **AUTH - PostSignUp:** Registrar un usuario en el sistema.
  `(POST) /users/`

Datos del formulario para x-www-form-urlencoded

Campo	Tipo	Descripción
username	*Text*	Nombre de usuario
email	*Text*	Email
password	*Text*	Contraseña
profile.lang.code	*Text*	Código de idioma. Actualmente los valores posibles son: *es*: Español *en*: Inglés

Respuesta


```
{
  "status": 201,
  "user": {
    "id": 9,
    "username": "movie",
    "email": "movie@mooviest.com",
    "password":
"pbkdf2_sha256$20000$wiNMenJqVebu$DUTwrahkp1n+teQZbnhGrcWI+e+8t
oKN6hC20xz5QgQ=",
    "profile": {
      "lang": {
        "code": "es"
      },
      "avatar": "/media/user/default/no-image.png"
    }
  },
  "errors": null,
  "token": "8a0ac632536d7d1ac5db90f6f05338cef2778516"
}
```


Obtenemos los datos del usuario y su perfil, además del *token*, necesario para realizar el resto de peticiones que no son de tipo *AUTH* (Autorizadas).

Error


```
{
  "status": 400,
  "user": null,
  "errors": {
    "username": [
      "Ya existe un usuario con ese nombre de usuario."
    ],
    "email": [
      "Ya existe un usuario con ese email."
    ]
  },
  "token": null
}
```


Si ya existe un usuario en el sistema con los campos introducidos nos devolverá un campo de error con los mensajes correspondientes.

Nota: Para el resto de peticiones se utiliza un método de autenticación por *token* para dar seguridad a la API y que solo los usuarios registrados puedan realizar peticiones al resto de métodos. Se añadirá un campo de *Authorization* a la cabecera como el siguiente.

Authorization: Token {token}

Campo	Tipo	Descripción
token	*Text*	Token asociado a usuario del sistema

- **User - GetUserProfile:** Obtener el perfil de un usuario.
  `(GET) /users/{id}/`

  URL

Campo	Tipo	Descripción
id	*Number*	Id de un usuario registrado en el sistema

  Respuesta

```
{
  "status": 200,
  "user": {
    "first_name": "",
    "last_name": "",
    "username": "movie",
    "profile": {
      "born": null,
      "gender": null,
      "avatar": "/media/user/default/no-image.png",
      "lang": {
        "code": "es"
      },
      "postalCode": null,
      "city": null
    },
    "id": 9,
    "email": "movie@mooviest.com"
  }
}
```

  Obtenemos todos los datos del usuario y su perfil.

- **User - UpdateUserProfile:** Actualizar el perfil de un usuario.
  `(PUT) /users/{id}/`

  URL

Campo	Tipo	Descripción
id	*Number*	Id de un usuario registrado en el sistema

Datos del formulario para multipart/form-data

Campo	Tipo	Descripción
username	*Text*	Nombre de usuario
profile.lang.code	*Text*	Código de idioma
first_name (opcional)	*Text*	Nombre
last_name (opcional)	*Text*	Apellidos
email (opcional)	*Text*	Email
profile.born (opcional)	*Date*	Fecha de nacimiento
profile.city (opcional)	*Text*	Ciudad
profile.postalCode (opcional)	*Text*	Código postal
image (opcional)	*Image*	Imagen del perfil

Respuesta

Devolvería la misma respuesta que en la petición **GetUserProfile**

- **User - GetSwipeList:** Obtener la lista de películas para el sistema de clasificación (*swipe*), es decir, películas que no están clasificadas en ninguna lista del usuario.
  `(GET) /users/{id}/swipelist/`

  URL

Campo	Tipo	Descripción
id	*Number*	Id de un usuario registrado en el sistema

  Respuesta


```
{
  "count": 10,
  "next": null,
  "previous": null,
  "results": [
    {
      "movie_lang_id": 54625,
      "image": "EXTERNAL#pelis/HP9YTUHXX6",
      "title": "Cartas desde la locura",
      "collection": null,
```

```
      "id": 28689,
      "average": 0
    },
    {
      "movie_lang_id": 48254,
      "image": "EXTERNAL#pelis/VHRDRXK7X4",
      "title": "Ring of fire II: Sangre y acero",
      "collection": null,
      "id": 25274,
      "average": 0
    },
    {
      "movie_lang_id": 7581,
      "image": "/14/f3/14f30376096122243ff4fe3de8db0cb6.jpg",
      "title": "Uno de los nuestros",
      "collection": null,
      "id": 3813,
      "average": 0
    },
                .
                .
                .
  ]
}
```


Obtenemos la información necesaria de diez películas para mostrar en el sistema de clasificación (*swipe*). Nos devuelve su título en el idioma del perfil del usuario, la imagen (*cover*) de la película, la puntuación de nuestro sistema, su id asociada a nuestra base de datos y su tipo de colección que, en este caso, al ser para el sistema de clasificación y no estar aún clasificada, nos devolverá *null*.

- **User - GetCollectionList:** Obtener la lista de películas de un tipo concreto.
  `(GET) /users/{id}/collection/?name={name}&page={page}`

  URL

Campo	Tipo	Descripción
id	*Number*	Id de un usuario registrado en el sistema
name	*Text*	Nombre de la colección. Los tipos de colección son: seen, favourite, watchlist, blacklist
page	*Number*	Número de página

Respuesta


```
{
  "count": 174,
  "next": "/users/9/collection/?name=favourite&page=2",
  "previous": null,
  "results": [
    {
      "movie_lang_id": 2919,
      "image": "/a4/9d/a49dd25d5df38f2e97c4bfd7d4875684.jpg",
      "title": "El Señor de los Anillos: La Comunidad del
Anillo",
      "collection": {
        "typeMovie": "favourite",
        "id": 1
      },
      "id": 1468,
      "average": 0
    },
    {
      "movie_lang_id": 6140,
      "image": "/1d/8f/1d8f6066a9ad1cb91211bcbfca5915f2.jpg",
      "title": "Piratas del Caribe: La maldición de la Perla
Negra",
      "collection": {
        "typeMovie": "favourite",
        "id": 9
      },
      "id": 3088,
      "average": 0
    },
    {
      "movie_lang_id": 6136,
      "image": "/1c/3e/1c3e15050ceaf0e4edb15cdfb3c35867.jpg",
      "title": "Piratas del Caribe: El cofre del hombre
muerto",
      "collection": {
        "typeMovie": "favourite",
        "id": 13
      },
      "id": 3086,
      "average": 0
    },
                .
                .
                .

  ]
}
```

Obtenemos un resultado paginado con la información necesaria, para poder mostrar una previsualización de las películas. Nos devuelve su título en el idioma del perfil del usuario, la imagen (*cover*) de la película, la puntuación de nuestro sistema, su id asociada a nuestra base de datos y su tipo de colección.

- **User - PostMovieCollection:** Clasificar una película en la colección de un usuario determinada.
  `(POST) /collection/`

  Datos del formulario para x-www-form-urlencoded

Campo	Tipo	Descripción
user	*Number*	Id de un usuario del sistema
movie	*Number*	Id de una película del sistema
typeMovie	*Text*	Nombre de la colección. Los tipos de colección son: seen, favourite, watchlist, blacklist

  Respuesta

```
{
  "id": 854,
  "user": 9,
  "movie": 3088,
  "typeMovie": "favourite",
  "pub_date": "2016-11-06T07:44:48.756881Z"
}
```

  Obtenemos el resultado de haber clasificado una película en una colección determinada; la id de la tabla ***Collection***, la id del usuario, la id de la película, el tipo de la colección y la fecha de clasificación.

- **User - PatchMovieCollection:** Actualizar la colección de una película de un usuario.
  `(PATCH) /collection/{id}/`

  URL

Campo	Tipo	Descripción
id	*Number*	Id de la tabla ***Collection*** donde está clasificada la película que queremos actualizar

Datos del formulario para x-www-form-urlencoded

Campo	Tipo	Descripción
typeMovie	*Text*	Nombre de la colección. Los tipos de colección son: *seen, favourite, watchlist, blacklist*

Respuesta


```
{
  "id": 854,
  "user": 9,
  "movie": 3088,
  "typeMovie": "seen",
  "pub_date": "2016-11-06T07:48:47.107361Z"
}
```


Obtenemos el resultado de haber actualizado el tipo de colección de una película; la id de la tabla ***Collection***, la id del usuario, la id de la película, el tipo de la colección y la fecha de clasificación.

5.1.2.2 Movie

- **Movie - GetMovieDetail:** Obtener la información detallada de una película.
  `(GET) /movie/{id}/?movie_lang_id={movie_lang_id}&user_id={user_id}`

URL

Campo	Tipo	Descripción
id	*Number*	Id de la película a obtener
movie_lang_id	*Number*	Id de la tabla ***Movie_lang*** que identifica a la película en un idioma.
user_id	*Number*	Id del usuario que obtiene la película.

Respuesta


```
{
  "title": "Piratas del Caribe: La maldición de la Perla Negra",
  "participations": [
    {
```

```
    "celebrity": {
      "id": 6687,
      "name": "Terry Rossio",
      "born": null,
      "image": "/0c/c8/0cc8bb88f9135835849485bf8f27cf7b.jpg",
      "twitter_account": "",
      "address": ""
    },
    "role": "Escritor",
    "character": "",
    "award": ""
  },
  {
    "celebrity": {
      "id": 11741,
      "name": "Gore Verbinski",
      "born": null,
      "image": "/47/3b/473b2276bf3a902a427642afa76a87f8.jpg",
      "twitter_account": "",
      "address": ""
    },
    "role": "Director",
    "character": "",
    "award": ""
  },
  {
    "celebrity": {
      "id": 9390,
      "name": "Keira Knightley",
      "born": null,
      "image": "/ec/cf/eccf02146ae943544067b273437c487b.jpg",
      "twitter_account": "",
      "address": ""
    },
    "role": "Actor",
    "character": "Elizabeth Swann",
    "award": ""
  },
  {
    "celebrity": {
      "id": 12913,
      "name": "Orlando Bloom",
      "born": null,
      "image": "/f2/ce/f2cea9813f0850ef86cdd64660849247.jpg",
      "twitter_account": "",
      "address": ""
    },
    "role": "Actor",
    "character": "Will Turner",
```

```
      "award": ""
    },
    {
      "celebrity": {
        "id": 1020,
        "name": "Johnny Depp",
        "born": null,
        "image": "/f0/2c/f02cf77dbedcf318cedf4e436ebfbcb6.jpg",
        "twitter_account": "",
        "address": ""
      },
      "role": "Actor",
      "character": "Jack Sparrow",
      "award": ""
    }
                .
                .
                .
  ],
  "collection": {
    "typeMovie": "seen",
    "id": 854
  },
  "movie_lang_id": 6140,
  "movie_producer": "Walt Disney Pictures | Jerry Bruckheimer
Films",
  "synopsis": "El aventurero Capitán Jack Sparrow recorre las
aguas caribeñas. Pero sus andanzas terminan cuando su enemigo,
el Capitán Barbossa le roba su barco, el Perla Negra, y ataca
la ciudad de Port Royal, secuestrando a Elizabeth Swann, hija
del Gobernador. Will Turner, el amigo de la infancia de
Elizabeth, se une a Jack para rescatarla y recuperar el Perla
Negra. Pero el prometido de Elizabeth, Comodoro Norrington, les
persigue a bordo del HMS Dauntless. Además, Barbossa y su
tripulación son víctimas de un conjuro por el que están
condenados a vivir eternamente, y a transformarse cada noche en
esqueletos vivientes, en fantasmas guerreros.",
  "id": 3088,
  "genres": [
    {
      "name": "Fantasía"
    },
    {
      "name": "Acción"
    },
    {
      "name": "Aventura"
    }
  ],
```

```
  "runtime": 143,
  "original_title": "Pirates of the Caribbean: The Curse of the
Black Pearl",
  "released": 2003,
  "average": 0,
  "country": null,
  "backdrop": "/1d/8f/1d8f6066a9ad1cb91211bcbfca5915f2.jpg",
  "ratings": [
    {
      "name": "IMDb",
      "rating": 81,
      "count": 811554,
      "date_update": "2016-08-24"
    },
    {
      "name": "Tviso",
      "rating": 86,
      "count": 70896,
      "date_update": "2016-08-24"
    }
  ],
  "image": "/1d/8f/1d8f6066a9ad1cb91211bcbfca5915f2.jpg"
}
```


Obtenemos la información detallada de una película.

- **Movie - GetSearchMovies:** Obtener el resultado de buscar películas por su título original o por el título del idioma del usuario que busca las películas.
  `(GET) /movie_lang/?title={title}&code={code}&page={page}`

  URL

Campo	Tipo	Descripción
title	*Text*	Título de una película a buscar.
code	*Text*	Código de idioma. Actualmente los valores posibles son: *es*: Español *en*: Inglés
page	*Text*	Número de página

Respuesta


```
{
  "count": 6,
  "next": null,
  "previous": null,
  "results": [
    {
      "title": "Piratas del Caribe: El cofre del hombre
muerto",
      "average": 0,
      "collection": null,
      "movie_lang_id": 6136,
      "image": "/1c/3e/1c3e15050ceaf0e4edb15cdfb3c35867.jpg",
      "id": 3086
    },
    {
      "title": "Piratas del Caribe: En el fin del mundo",
      "average": 0,
      "collection": null,
      "movie_lang_id": 6138,
      "image": "/52/67/52678bde4104c39aed70938d1d82dc1d.jpg",
      "id": 3087
    },
    {
      "title": "Piratas del Caribe: La maldición de la Perla
Negra",
      "average": 0,
      "collection": {
        "typeMovie": "seen",
        "id": 854
      },
      "movie_lang_id": 6140,
      "image": "/1d/8f/1d8f6066a9ad1cb91211bcbfca5915f2.jpg",
      "id": 3088
    },
    {
      "title": "Piratas del Caribe: En mareas misteriosas",
      "average": 0,
      "collection": null,
      "movie_lang_id": 12013,
      "image": "/5c/a0/5ca0ed89b7f45de6c912029b20151400.jpg",
      "id": 6042
    },
    {
      "title": "Piratas del Caribe: Los hombres muertos no
cuentan cuentos",
      "average": 0,
      "collection": null,
      "movie_lang_id": 39934,
```

```
      "image": "/15/09/15092393d1f215f1769a50d5f97118fa.jpg",
      "id": 20813
    },
    {
      "title": "Lego Pirates of the Caribbean: The Video Game",
      "average": 0,
      "collection": null,
      "movie_lang_id": 74761,
      "image": "EXTERNAL#pelis/WN5C4474VZWTCUN",
      "id": 40053
    }
  ]
}
```


Obtenemos la información necesaria de una o varias películas para mostrar una previsualización en un listado. Nos devuelve su título en el idioma seleccionado, la imagen (*cover*) de la película, la puntuación de nuestro sistema, su id asociada a nuestra base de datos y su tipo de colección.

5.2 Scripts

Parte de nuestra problemática era la complejidad del proceso de recopilación de datos de películas y su actualización automática accediendo a las *APIs* externas o páginas web de puntuaciones. Más concretamente, el principal problema es que hay cierta información, como las puntuaciones de las películas, que no es posible obtenerla a través de APIs desarrolladas por terceros. Por ello hemos desarrollado durante gran parte del proyecto un módulo consistente en un conjunto de *scripts* y *scrappers* que nos permite realizar esta labor.

Para la recopilación de datos, hemos creado un script principal, ***main_script.py,*** que se encarga de llamar a todos los demás scripts separados por funcionalidad y plataforma. La ***Figura 5.1*** muestra la estructura del proyecto.

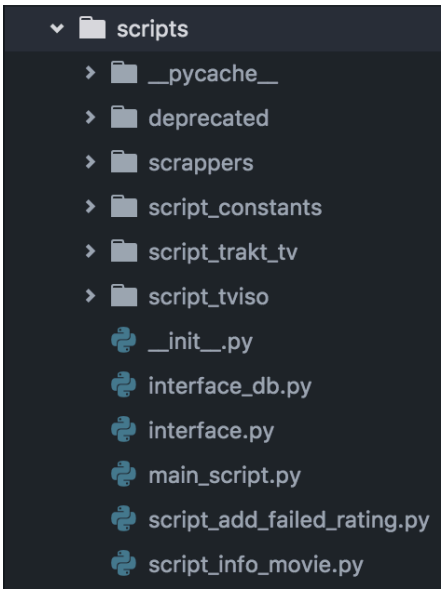


Figura 5.1 Estructura de carpetas de los scripts

Por otra parte, tenemos ***interface_db.py*** e ***interface.py***. El primero se encarga de crear y gestionar el contenido de la base de datos a través de una clase ***DB,*** que es inicializada en el script principal mediante su constructor, y el segundo se encarga de guardar el estado del programa y *logs* de ejecución, además de enviar *emails* con dicha información.

Haciendo uso de paquetes y librerías *Python* como *"http.client"* creamos el cliente *HTTP* que permite realizar peticiones a la base de datos, como podemos apreciar en el siguiente fragmento de código.

```
def __init__(self,token):
      # Generación del cliente HTTP y autorización por Token
      self.connection =
http.client.HTTPConnection("127.0.0.1",8000)
      self.headers = { "Authorization" : "Token " + token,
                  "Content-type": "application/json"}
```

Mediante el paquete *"urllib.request"* realizamos peticiones a la *API*, con los diferentes métodos *HTTP*. Para el caso de métodos *POST* o *PUT*, enviamos los datos en formato *JSON* y en todos los casos recibimos la respuesta de la petición en este mismo formato.

```
def insert_data(self, api_url, js):
     self.connection.request('POST', api_url, js, self.headers)
     res = self.connection.getresponse()
     data = res.read().decode("utf8")
     return json.loads(data)
```

Como vemos en este fragmento de código, se trata de una petición *POST*, donde le pasamos la *URL* de la *API* a donde realizar la petición y los datos en formato *JSON*.

Volviendo al script principal, realizamos una fase de recogida de constantes en la base de datos, que solo realizamos la primera vez que se ejecuta el programa. Se guarda información como géneros, idiomas, países…

En las siguientes fases del script obtenemos toda la información detallada de una película a través de *APIs* de terceros. En concreto, para obtener la información en español, utilizamos Tviso y, para inglés, Trakt.tv. Por ello, como podemos observar en la ***Figura 5.2*** y la ***Figura 5.3***, hemos separado las funcionalidades y plataformas en carpetas para que el código quede más organizado.

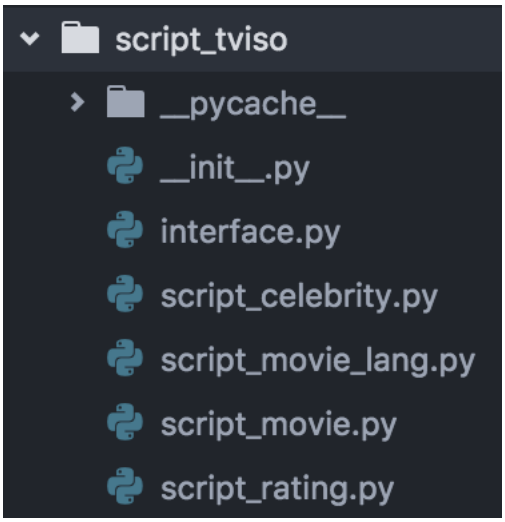


Figura 5.2 Estructura de carpetas de los scripts utilizados para la plataforma Tviso

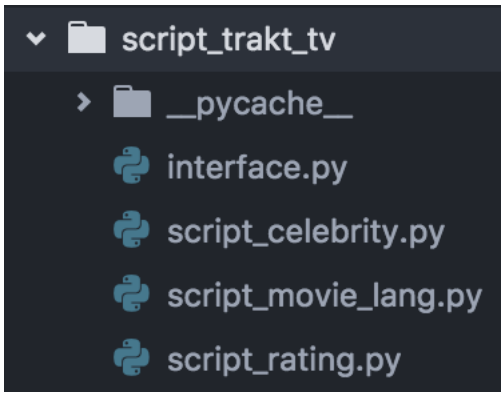


Figura 5.3 Estructura de carpetas de los scripts utilizados para la plataforma Trakt.tv

Otra parte sería la recogida de información que no nos ofrecen *APIs* externas, como son las puntuaciones de películas de algunas plataformas. Para ello hemos utilizado *scrappers* o arañas web. Los *scrappers* son herramientas software que rastrean y extraen cierta información de una página concreta. Haciendo uso de la librería *BeautifulSoup,* podemos obtener el código *HTML* de una página concreta, y mediante métodos de la librería como *find()* buscar la etiqueta *HTML* donde se encuentra la información que necesitemos.

```
# get_audience_rating(soup), get audience rating of Rotten
Tomatoes,
#   Params
#       - soup, page from BeautifulSoup
def get_audience_rating(soup):

    # AUDIENCE SCORE - RottenTomatoesAudience
    # Div with RottenTomatoesAudience
    lista = soup.find_all("div", {"class":"audiencepanel"})[0]
    # Rating
    rating = lista.find_all("div", {"class":"audience-score
meter"})[0].find("span","superPageFontColor").get_text()
    rating = int(rating.replace("%", ""))
    # Count
    count = lista.find("div","audience-
info").find_all("div")[1].get_text()
    count = int(count.replace("User Ratings:",
"").strip().replace(",",""))

    return rating, count
```

Como vemos en el fragmento de código anterior, buscamos la puntuación y el número de votos de una película en la página *RottenTomatoes*; en concreto la puntuación de la audiencia. Dependiendo de la estructura del código *HTML*, será más o menos complejo obtener una información concreta.

En este caso se acceden a varias clases y etiquetas. Por un lado, extraemos la puntuación de la clase *audience-score-meter* y por otro, el número de votos de la clase *audience-info*, que finalmente se devuelven como resultado de la función *get_audience_rating()*.

5.3 Creación e instalación del proyecto

La creación e instalación del proyecto se podría separar en el proyecto del servidor y el de la aplicación o cliente *Android*. Mostraremos cómo sería la creación de un proyecto desde cero, y cómo hacerlo a partir de nuestro repositorio privado en GitHub o teniendo ya la carpeta del mismo.

5.3.1 Aplicación Mooviest (Servidor Django)

En esta sección se explicará el proceso a realizar para crear e instalar el proyecto del servidor. Para ambos partimos de unos requisitos previos que veremos a continuación. Los comandos expuestos son para la instalación en un ordenador con sistema operativo MacOS o Linux. Más concretamente, para el desarrollo de este proyecto se ha utilizado un entorno MacOS.

En primer lugar, debemos tener instalado *Python* y su gestor de paquetes de ***pip***.

Una vez instalado, procederemos a preparar nuestro entorno virtual (*virtualenv*), que nos permite aislar la configuración de los paquetes instalados para cada proyecto, por lo que podemos trabajar, por ejemplo, con diferentes versiones de *Django* o *Python* en proyectos diferentes y en el mismo ordenador.

Para la instalación de *virtualenv* ejecutaremos el siguiente comando:

```
pip3 install virtualenv
```

Después crearemos nuestro directorio, donde instalaremos el entorno *virtualenv*.

```
mkdir mooviest

cd mooviest

python3 –m venv venv       # Creación entorno virtual

source venv/bin/actívate   # Activar entorno virtual
```

Como vemos en el fragmento anterior creamos y activamos el entorno virtual en el directorio del proyecto y, a partir de ahora, todo lo que instalemos mediante el comando *pip*, estará instalado de forma aislada en él.

5.3.1.1 Creación del proyecto

Una vez instalados todos los requisitos, podemos centrarnos en crear e instalar nuestro proyecto *Django* y añadir todos los módulos y paquetes necesarios. En primer lugar, instalaremos la versión de *Django* y crearemos nuestro proyecto.

```
pip install django==1.8

django-admin startproject mysite
```

El último comando nos crea la estructura de un proyecto *Django* con sus archivos y directorios por defecto, que podemos ampliar creando nuestras propias apps como haremos a continuación para comenzar el proyecto.

```
python3 manage.py startapp movie
```

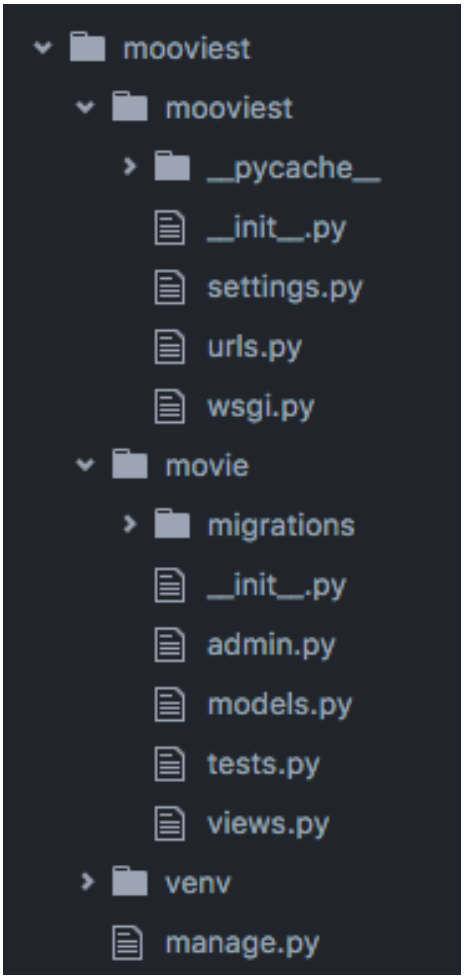


Figura 5.4 Estructura de carpetas de un proyecto Django

En *Django* se desarrolla creando aplicaciones en su propio directorio como podemos ver en la ***Figura 5.4***, donde acabamos de crear la aplicación *movie*. Además, tenemos un subdirectorio del raíz, *mooviest*, igual que el proyecto, que es donde se coordinan todas las apps mediante el archivo urls.py y donde se configuran sus

ajustes en settings.py; es por ello que al crear una app debemos añadirla a este archivo.

5.3.1.2 Instalación y configuración del proyecto

En este apartado veremos cómo instalar y configurar nuestro proyecto, desde nuestro repositorio privado en GitHub.

```
git clone https://github.com/JoseAntpr/mooviest.git
```

En este momento, ya tendremos nuestro proyecto en el directorio mooviest y nuestro entorno virtual como explicamos anteriormente. Tan solo debemos instalar las dependencias que existen en el archivo *requirements.txt*, mediante el siguiente comando.

```
pip3 install –r requirements.txt
```

5.3.1.3 Configuración de la base de datos

Para el correcto funcionamiento de nuestro proyecto, debemos tener instalada y configurada nuestra base de datos en PostgreSQL. Para ello comenzamos instalando *postgresql* con el siguiente comando:

```
brew install postgresql
```

Una vez hecho, procedemos a crear nuestra base de datos y un usuario:

```
createdb mooviest

createuser –P

psql

>> GRANT ALL PRIVILEGES ON DATABASE mooviest TO root;
```

Como comentamos anteriormente, la configuración del proyecto *Django* está en el archivo *settings.py*, entonces, una vez creamos la base de datos, debemos añadir su configuración a este archivo.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mooviest',
        'USER': 'root',
        'PASSWORD': 'root',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Por último, ejecutamos las migraciones de la base de datos y las ejecutamos con los siguientes comandos:

```
python3 manage.py makemigrations #Creación de las migraciones

python3 manage.py migrate #Ejecución de las migraciones
```

Cada vez que queramos realizar una modificación en el modelo de nuestra base de datos o crearlo por primera vez, debemos ejecutar las migraciones del proyecto. Las migraciones, son el código necesario que *Django* utiliza para realizar el cambio del modelo de nuestra base de datos, como, por ejemplo, la modificación del tipo de un atributo, la creación de una nueva tabla y sus atributos, etc. Se crean, si *Django* detecta que hay un cambio entre su modelo interno y el de nuestra base de datos.

Una vez han sido generadas con el primer comando del código anterior, se ejecutan con el segundo comando, en este momento, *Django* se encarga de ejecutar las sentencias necesarias en *PostgreSQL* para realizar la modificación en la base de datos.

Finalmente, para iniciar nuestro proyecto, ejecutaremos el siguiente comando:

```
python3 manage.py runserver 0.0.0.0:8000
```

Al introducir la *IP 0.0.0.0*, podremos recibir peticiones en nuestra máquina desde cualquier *IP* de la red local.

5.3.2 Aplicación Mooviest (Android)

En este apartado se explicará el proceso a realizar para instalar y configurar nuestro proyecto Android. Los requisitos previos son los siguientes:

- **IDE de desarrollo Android Studio.** Tener instalado el IDE de desarrollo Android Studio, que podemos descargarlo desde su página oficial (https://developer.android.com/studio/index.html).

- **Android SDK.** Tener instalado el conjunto de herramientas de desarrollo de aplicaciones Android. Es recomendable descargar el paquete de Android Studio donde viene integrado el Android SDK.

- **Min SDK 19.** Instalar el SDK 19 que se ha establecido como la mínima versión requerida para poder ejecutar el proyecto en un dispositivo Android. En este caso el SDK 19 corresponde a la versión 4.4 de Android también llamada Kit Kat.

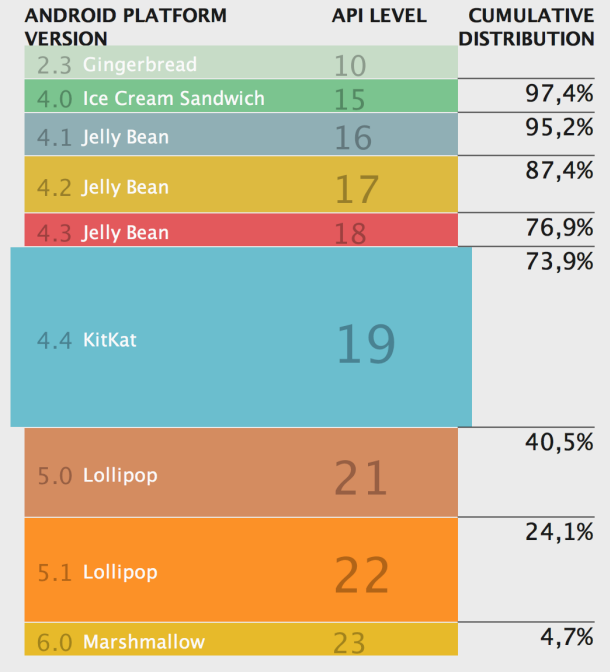


Figura 5.5 Porcentaje de dispositivos activos en Google Play Store que utilizan una versión de Android determinada

Se ha escogido esta versión como la mínima requerida porque, como podemos apreciar en la ***Figura 5.5,*** el 73,9% de los dispositivos actuales podrían ejecutarla. Con lo cual abarca a un gran número de usuarios.

- **JDK (Java Development Kit).** Tener instalado el conjunto de herramientas de desarrollo para la creación de programas en Java, accesible desde la página de Oracle (https://www.oracle.com/index.html).

5.3.2.1 Creación del proyecto

Una vez cumplidos los requisitos anteriormente citados, procederemos a crear nuestro proyecto *Android*. En primer lugar, abrimos nuestro *IDE* de desarrollo *Android Studio* y seleccionamos la opción de empezar un nuevo proyecto *Android Studio*.

Seleccionamos el nombre de la aplicación, en este caso *Mooviest* y el *SDK* mínimo para la ejecución del proyecto, elegimos *API 19 Android 4.4 (KitKat).* Creamos la actividad principal *MainActivity* y pulsamos en finalizar.

En este momento tendremos la estructura del proyecto y los archivos y directorios por defecto que nos crea Android Studio, como podemos apreciar en la ***Figura 5.6.***

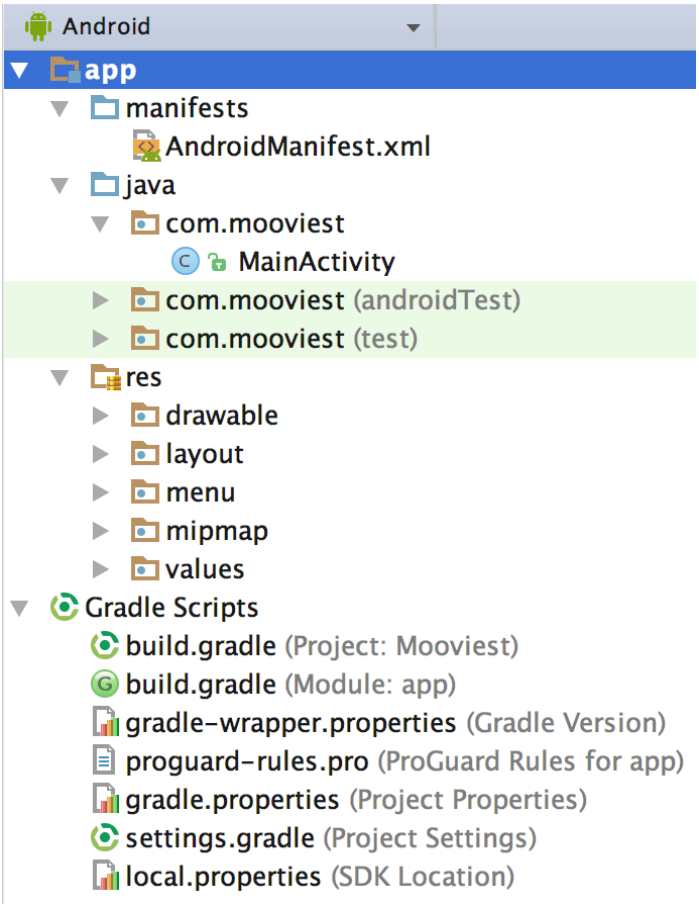


Figura 5.6 Estructura de carpetas de un proyecto Android

A continuación, explicaremos cómo funciona una aplicación *Android* y la estructura de carpetas de un proyecto.

Una aplicación *Android* está basada en una serie de actividades. Las actividades (*activities*), representan el componente principal de la interfaz gráfica de una aplicación *Android*. Se puede pensar en una actividad, como el elemento análogo a una ventana o pantalla en cualquier otro lenguaje visual. Al fin y al cabo, una actividad se trata de una clase *Java*, que contiene todo el código que se quiere implementar en la aplicación. Una *activity*, siempre está relacionada con un fichero *XML*, que contiene el *layout*, vista o parte gráfica.

La configuración del proyecto *Android* se encuentra en el archivo *AndroidManifest.xml*, que contiene las versiones de la aplicación, la definición de las *activities*, y los permisos de la aplicación para acceder al dispositivo como, por ejemplo, el acceso a Internet, a la lectura y escritura de la memoria interna del dispositivo, el acceso a la cámara…

Una ventaja de utilizar Android Studio frente a otros *IDE* es la de poder hacer uso de ***gradle***, que nos permite manejar fácilmente el uso de dependencias del proyecto o tener varios entornos para un mismo proyecto entre otros. En un proyecto con *gradle* nos encontramos dos archivos importantes que son ***settings.gradle*** y

build.gradle. El primero organiza los módulos de nuestra aplicación, tanto el proyecto principal, conocido como *app,* como las librerías que hubiésemos creado. El segundo describe el proceso del ***build*** del módulo: las versiones del *SDK* que se deben usar, las dependencias, la forma de generar nuestro *apk*, las especificaciones del lenguaje… Cada módulo de un proyecto cuenta con su propio archivo *build.gradle*.

Por último, veremos cómo está organizado y para qué sirven las carpetas más importantes de un proyecto Android:

- **Carpeta /src/:** esta carpeta contendrá todo el código fuente de la aplicación, todos los recursos *XML*, *activities*, clases auxiliares…
- **Carpeta /java/:** contiene todas las *activities* y clases auxiliares de nuestro proyecto, organizado en subcarpetas dependiendo de la funcionalidad.
- **Carpeta /res/:** contiene todos los ficheros de recursos necesarios para el proyecto: imágenes, traducciones, interfaces gráficas *XML*…
- **/res/drawable/:** contiene las imágenes o iconos de nuestra aplicación en varias calidades y resoluciones.
- **/res/layout/:** contiene los ficheros de definición *XML* de las diferentes pantallas de la interfaz gráfica.
- **/res/values/strings:** contiene todas las cadenas de texto y traducciones de la aplicación.

5.3.2.2 Instalación y configuración del proyecto

En este apartado, veremos cómo instalar el proyecto desde nuestro repositorio privado en ***GitHub***.

```
git clone https://github.com/JoseAntpr/mooviest_android.git
```

En este momento tendremos la carpeta del proyecto, que solo tendremos que abrir con nuestro *IDE* de desarrollo *Android Studio* y poder ejecutarlo en el emulador que nos proporciona el *IDE*, o en un dispositivo Android con el modo *"Depuración USB" habilitado y* conectado mediante un cable USB.

5.3.2.3 GitHub

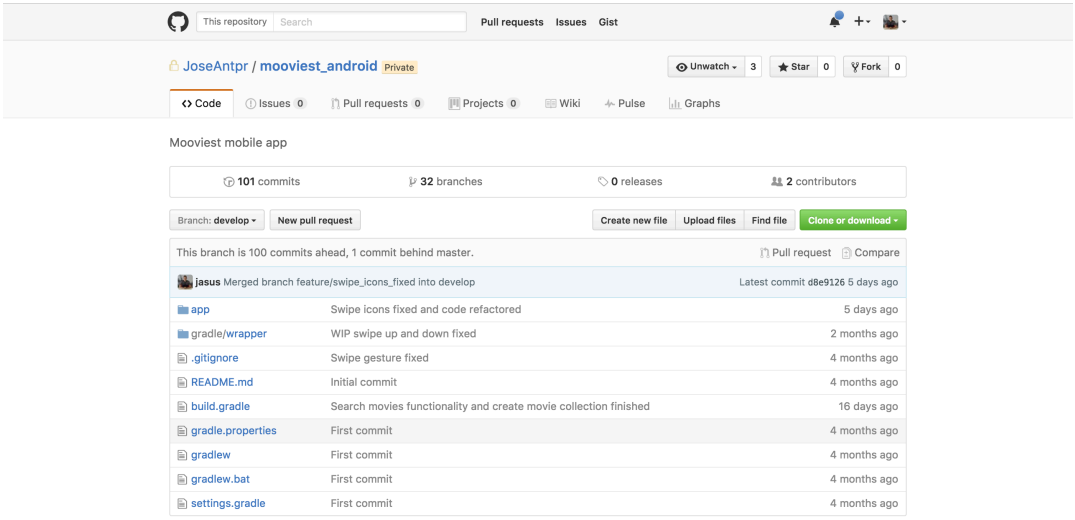


Figura 5.7 Proyecto mooviest de la aplicación Android en la plataforma GitHub

A lo largo de la realización del proyecto se ha utilizado junto con ***Git*** como *VCS* (sistema de control de versiones) y, de esta forma, tener así controladas todas las *features* (pequeñas funcionalidades del desarrollo del proyecto) y versiones del proyecto.

5.4 Desarrollo de la aplicación Mooviest (Android)

5.4.1 Introducción, inicio de sesión y registro

Figura 5.8 Pantalla inicial (Splash screen)

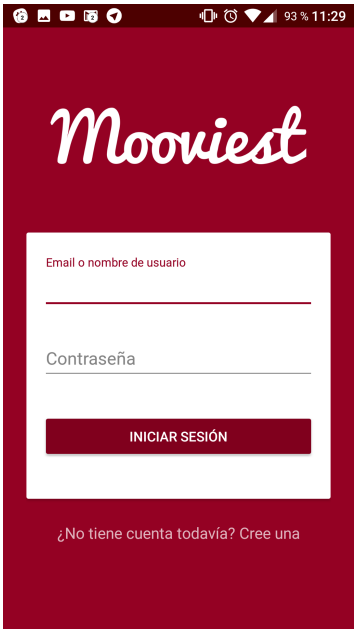


Figura 5.9 Pantalla para iniciar sesión en el sistema

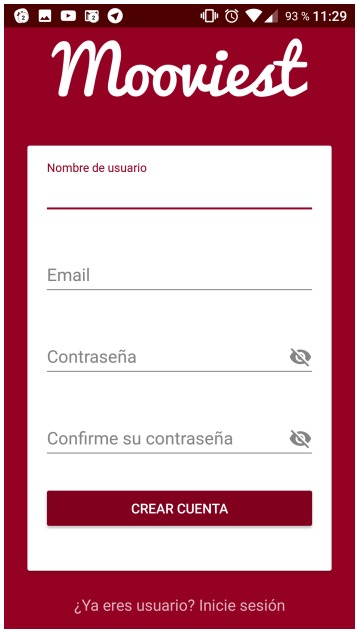


Figura 5.10 Pantalla para registrarse en el sistema

Nada más abrimos la aplicación veríamos la pantalla de introducción o *splash screen* correspondiente a la ***Figura 5.8***. Esta pantalla corresponde a la *activity* ***SplashScreenActivity.java*** de nuestro proyecto Android.

En esta *activity* se realizan las primeras decisiones de la aplicación. En primer lugar, se obtienen o inicializan las variables que se guardarán en la memoria del dispositivo del usuario, dependiendo de si es el primer inicio de la *app* o no.

```
SharedPreferences app_prefs = getSharedPreferences("APP_PREFS",
Context.MODE_PRIVATE);
boolean tutorial = app_prefs.getBoolean("tutorial", false);
boolean logged = app_prefs.getBoolean("logged", false);

SharedPreferences user_prefs = getSharedPreferences("USER_PREFS",
Context.MODE_PRIVATE);
user_prefs.getBoolean("default_avatar", true);
user_prefs.getString("avatar_image", "");
user_prefs.getInt("id", 0);
user_prefs.getString("username", "");
user_prefs.getString("email", "");
user_prefs.getString("token", "");
```

Como vemos en este fragmento de código, las identificamos como ***APP_PREFS*** o preferencias de la aplicación, para saber si un usuario ya ha completado el tutorial de la aplicación o ya ha iniciado sesión en ella. ***USER_PREFS*** o preferencias de usuario, se emplea para guardar en la memoria del dispositivo la *id* del usuario que ha iniciado sesión, su nombre de usuario, *email*, *token, etc.* Este último necesario para realizar las peticiones a la API de manera autenticada una vez el usuario está identificado en la *app*.

Una vez obtenidas o inicializadas estas variables, comprobamos si el usuario ya ha iniciado sesión en la *app*. En caso afirmativo, comprobamos si el dispositivo tiene conexión a internet y se redirige automáticamente a la pantalla *home* o principal de la app ***Figura 5.18.*** En caso de que no haya iniciado sesión en la *app* se le redirigirá a la pantalla para iniciar sesión ***Figura 5.9***.

En la pantalla para iniciar sesión, podremos introducir el email o nombre de un usuario registrado en el sistema, y su contraseña asociada, para así, iniciar sesión en el sistema.

Además, disponemos de un enlace a la pantalla para que un usuario pueda registrarse en el sistema, correspondiente a la ***Figura 5.10***. Contendrá los campos obligatorios como son nombre de usuario, *email*, contraseña y otro campo para volver a introducir la contraseña, que nos servirá para validarla con la anterior, antes de enviar al sistema el registro. De esta forma, el usuario tendrá menos posibilidad de

haber introducido una contraseña no deseada. Además, se han validado los campos de email, para no permitir introducir un email no válido; nombre de usuario, para que al menos tenga 3 caracteres; y la contraseña, que al menos debe contener 6 caracteres alfanuméricos, como se puede apreciar en la ***Figura 5.11***. En los campos de contraseña, permitimos al usuario la opción de mostrarla u ocultarla para comprobar que no se ha equivocado al introducirla.

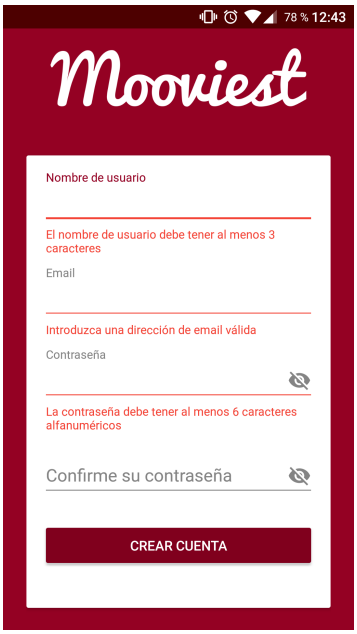


Figura 5.11 Errores en el resgistro de un usuario

También, para evitar el registro de usuarios con el mismo nombre de usuario o email, realizamos dicha comprobación en el servidor y mostramos el error correspondiente al usuario, tal y como podemos ver en la ***Figura 5.12***.

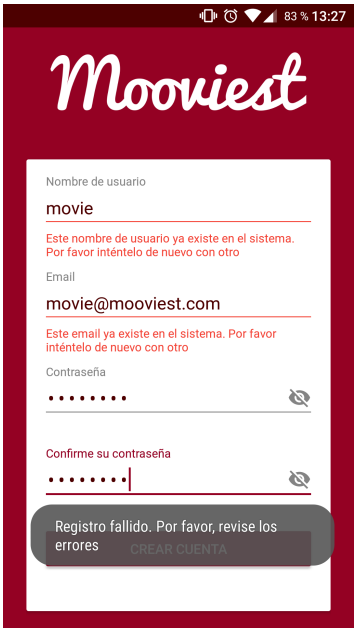


Figura 5.12 Error en el registro de un usuario. El email y el nombre de usuario ya existen en el sistema.

Tanto la opción de iniciar sesión como registrarse, son peticiones a la *API* que no necesitan autenticación por *token*. Una vez realizada una u otra, como vimos en las peticiones de usuario a la *API*, en la respuesta nos devolvían los datos del usuario, que guardaremos en las preferencias de la app y usuario que hemos citado anteriormente, para que, al volver a entrar en la aplicación, estemos autenticados en el sistema. Por último, el usuario será redirigido a la pantalla *home* de la *app*.

5.4.2 *Home*

La pantalla de *home* es la principal de la aplicación y la que se ve cuando el usuario ha iniciado sesión en el sistema o se ha registrado. Contiene varios apartados que son los siguientes:

- Menú lateral para el acceso a todas las secciones de la aplicación.
- Sistema de clasificación de películas.
- Listas del usuario
- Botón para la búsqueda de películas.

En nuestro proyecto Android se identifica con la *activity* ***HomeActivity.java***. En ella se crean el menú lateral (***DrawerLayout***), la barra superior (***TabLayout***) donde se encuentra el botón de búsqueda y las páginas del sistema de clasificación y listas de usuario, con el uso de un ***ViewPager***, que nos permite crear una vista de *tabs,* donde podemos desplazarnos a derecha o izquierda por los *tabs*. Las pantallas del sistema de clasificación y las listas serían ***Fragments*** dentro del ***ViewPager*** citado anteriormente.

```
@Override
protected void onCreate(Bundle savedInstanceState)
                .
                .
                .
   viewPager = (ViewPager) findViewById(R.id.viewpager);
   //GUARDA EL ESTADO DE TODOS LOS FRAGMENTS DEL VIEW PAGER
   viewPager.setOffscreenPageLimit(2);
   //SETUP VIEW PAGER (ADD 2 FRAGMENTS)
   setupViewPager(viewPager);
                .
                .
                .
}


private void setupViewPager(ViewPager viewPager) {
```

```
    //CREACIÓN DEL VIEW PAGER
        ViewPagerAdapter adapter = new
    ViewPagerAdapter(getSupportFragmentManager());
    //FRAGMENT DEL SISTEMA DE CLASIFICACIÓN
        adapter.addFragment(new SwipeFragment(),
    getString(R.string.swipe));
    //FRAGMENT DE LAS LISTAS DEL USUARIO
        adapter.addFragment(new UserListsFragment(),
    getString(R.string.my_lists));
        viewPager.setAdapter(adapter);
}
```

Como podemos ver en el siguiente código, al crear la *activity* ***HomeActivity.java*** en el método ***onCreate()***, obtenemos el ***ViewPager*** de la vista y lo creamos en el método ***setupViewPager()*** añadiéndole los dos ***Fragment*** para la vista del sistema de clasificación (***SwipeFragment.java***) y la vista de las listas del usuario (***UserListsFragment.java***).

5.4.3 Menú lateral

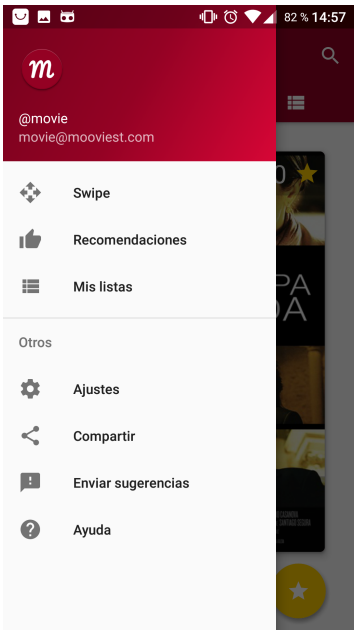


Figura 5.13 Menú lateral de la aplicación

Como podemos ver en la ***Figura 5.13,*** desde el menú lateral tenemos acceso a todas las secciones de la *app*. Podemos acceder a los diferentes *tabs* de la app, como son, '*Swipe'* y *'Mis listas'*.

Además, al perfil del usuario como se ve en la parte superior de éste. Haciendo clic en esa sección nos redirigirá a la *activity* del perfil del usuario, identificada en nuestro proyecto Android como ***ProfileActivity.java***.

5.4.4 Perfil del usuario

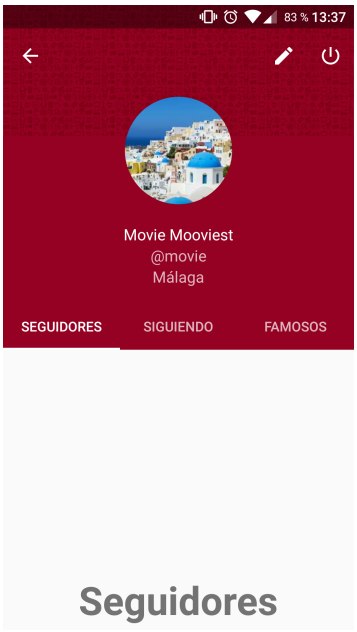


Figura 5.14 Perfil de un usuario

En esta vista, correspondiente a la ***Figura 5.14,*** tendremos la opción de ver el perfil del usuario, editar su perfil o cerrar sesión en la app. Estas dos últimas opciones están disponibles como podemos ver en la barra superior.

La acción de cerrar sesión resetea todas las preferencias del usuario (***USER_PREFS***) del dispositivo, y redirecciona al usuario a la pantalla para iniciar sesión de nuevo.

La acción de editar el perfil del usuario redirige a la pantalla de edición de perfil.

5.4.5 Editar perfil del usuario

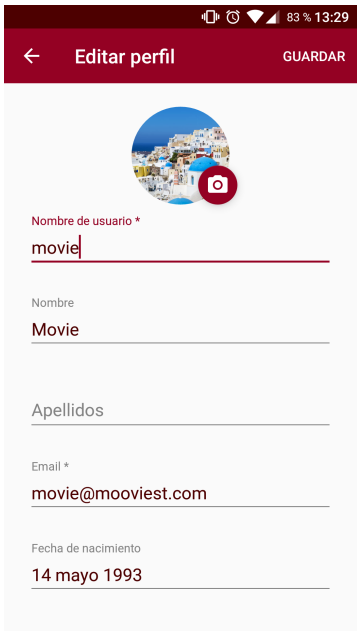


Figura 5.15 Pantalla de edición del perfil de un usuario

Figura 5.16 Calendario para seleccionar la fecha de nacimiento de un usuario

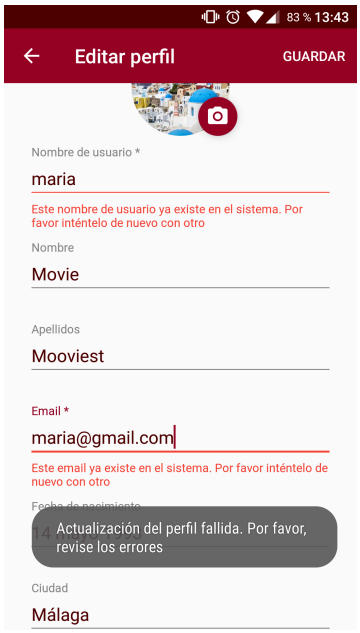


Figura 5.17 Error en la actualización del perfil de un usuario

Corresponde con la *activity* ***EditProfileActivity.java*** y en ella podemos editar el perfil del usuario. Como vemos en la ***Figura 5.15*** y la ***Figura 5.16,*** podemos editar su imagen de perfil, nombre de usuario, nombre, apellidos, email, fecha de nacimiento, género, ciudad y código postal.

Para actualizar la imagen del perfil, damos acceso a que el usuario obtenga la imagen de su galería, como podemos ver en este fragmento de código.

```
Intent takePictureIntent = new Intent(Intent.ACTION_PICK,
        MediaStore.Images.Media.EXTERNAL_CONTENT_URI);
if (takePictureIntent.resolveActivity(getPackageManager()) !=
null) {
    startActivityForResult(takePictureIntent,
REQUEST_IMAGE_CAPTURE);
}
```

Si ha seleccionado una, la mostramos para que el usuario sepa cómo quedaría en su perfil antes de guardarlo.

Asimismo, como hicimos en el registro de un usuario, comprobamos si existe en el sistema otro usuario con el mismo email o nombre de usuario; en ese caso mostramos dicho error, tal y como vemos en la ***Figura 5.17***.

5.4.6 Sistema de clasificación de películas (*Swipe*)

Figura 5.18 Sistema de clasificación de películas y pantalla principal

Volvamos a la vista principal de la app, concretamente al sistema de clasificación de películas (*swipe*), ***Figura 5.18***. En ella podemos ver la carátula de una película, junto con su título, en el idioma del dispositivo del usuario y la puntuación de la película en nuestro sistema y cuatro botones para clasificarla en las cuatro listas disponibles. Además de estos botones, también tenemos la posibilidad de hacer *swipe* sobre la película, es decir, mediante gestos podríamos clasificarla.

- **Botón rojo (Cruz) o *swipe* hacia abajo:** La clasifica en la lista negra o *blacklist* del usuario, es decir, no le interesan al usuario.
- **Botón azul (Marcapáginas) o swipe hacia la izquierda:** La clasifica en la lista de películas pendientes del usuario.
- **Botón verde (Ojo) o swipe hacia la derecha:** La clasifica en la lista de películas vistas por el usuario.
- **Botón amarillo (Estrella) o swipe hacia arriba:** La clasifica en la lista de películas favoritas del usuario.

El control de gestos mediante *swipe*, se ha llevado a cabo, definiendo una interfaz entre el gestor de movimientos y la vista, por tanto, cuando el gestor de movimiento detecta que se ha movido una película, la interfaz notifica a la vista, con el gesto que se ha realizado, y así poder tomar la decisión de clasificarla en una lista u otra.

Para la realización de esta vista utilizamos un objeto ***Adapter***, que se encarga de crear una vista y repetirla para cada *ítem,* de una colección pasada por parámetro, en este caso para las películas.

De este modo, hemos definido un *array* inicial de diez películas a modo de *buffer*, que sería lo que nos devuelve la petición de la *API* para obtener las películas del *swipe*. En el *adapter* no introducimos el *array* completo de películas, sino que vamos introduciendo de dos en dos, es decir, en la vista habría siempre dos películas, la que se ve y una detrás de ésta. Cuando clasificamos la película que vemos en el *swipe*, esta desaparece de la vista dando paso a la siguiente, en este momento se introduce en el *adapter* la siguiente película del buffer, y de este modo siempre tendríamos dos películas en el *adapter* y en concreto en la vista.

Además de añadir una nueva, comprobamos si el *buffer* de películas tiene menos de seis. En este caso, añadiríamos al *array* diez películas más, llamando a la petición de la *API*. En este momento tendríamos las cinco películas que había, más otras diez nuevas que hemos añadido y, de esta forma, continuamente cargaríamos películas en esta vista.

```
adapter = new SwipeDeckAdapter(movies_swipe, getContext(), this);
cardStack.setAdapter(adapter);

cardstack.setEventCallback(new SwipeDeck.SwipeEventCallback() {
    @Override
    public void cardSwipedLeft(int position) {
        movieCollectionTask("watchlist", adapter.getItem(0));
        checkMoviesSwipe();
        removeAndAddMovieToAdapter(0, 1);
    }

    @Override
    public void cardSwipedRight(int position) {
        movieCollectionTask("seen", adapter.getItem(0));
        checkMoviesSwipe();
        removeAndAddMovieToAdapter(0, 1);
    }

    @Override
    public void cardSwipedUp(int position) {
        movieCollectionTask("favourite", adapter.getItem(0));
        checkMoviesSwipe();
        removeAndAddMovieToAdapter(0, 1);
    }

    @Override
    public void cardSwipedDown(int position) {
        movieCollectionTask("blacklist", adapter.getItem(0));
        checkMoviesSwipe();
        removeAndAddMovieToAdapter(0, 1);
    }

    @Override
    public void cardRemove(int position) {
```

```
            checkMoviesSwipe();
            removeAndAddMovieToAdapter(0, 1);
        }

});
```

En este fragmento vemos cómo sería el proceso para clasificar una película:

1. Realizamos la petición a la *API* que nos añade la película a la colección seleccionada, con el método *movieCollectionTask()*.
2. Comprobamos si el *buffer* de películas para el *swipe* tiene menos de seis para añadir más a éste.
3. Eliminamos del *adapter* la película que acabamos de clasificar y añadimos una nueva detrás de la última.

5.4.7 Listas del usuario

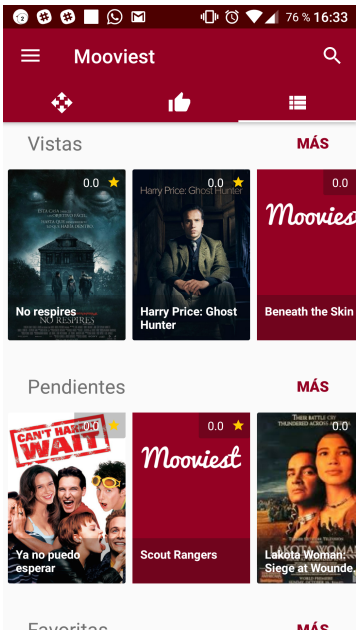


Figura 5.19 Listas clasificadas por el usuario

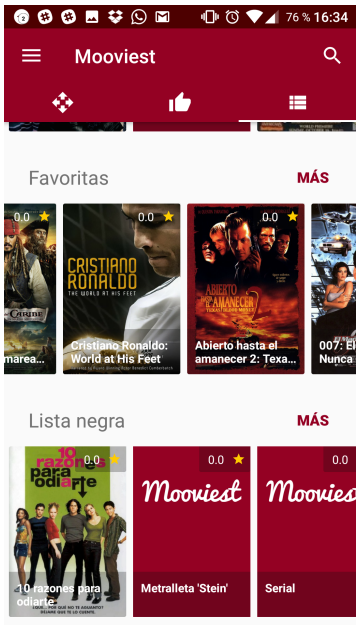


Figura 5.20 Listas clasificadas por el usuario

En esta vista ***Figura 5.19*** y ***Figura 5.20,*** mostramos las cuatro listas posibles en las que el usuario puede clasificar una película. A modo de visión general, muestra cuáles son las últimas películas que el usuario ha clasificado en estas listas. Además, el usuario tiene la opción de pulsar sobre una y ser redirigido a la vista de detalle, o ver una lista completa pulsando sobre el botón '*MÁS'.*

Para su desarrollo hemos utilizado el mismo sistema que para la vista del *swipe*, es decir, cuatro *adapters* que contienen un *array* de diez películas cada uno. En esta vista el *adapter* lo hemos configurado para que permita realizar un *scroll* horizontal, y

así ver las demás portadas sin que ocupe demasiado espacio en la pantalla verticalmente.

5.4.8 Lista completa

Figura 5.21 Lista completa de películas favoritas de un usuario

Como hemos dicho anteriormente, el usuario tiene la opción de ver una lista completa de entre sus cuatro posibles, como podemos ver en la ***Figura 5.21***, la lista de películas favoritas. En ella podemos pulsar sobre una y ser redirigidos a su vista de detalle.

Vamos mostrando de diez en diez y tres películas por fila y, si el usuario quiere seguir viendo más, cuando vaya haciendo *scroll* hacia abajo, se cargarán otras diez por la cola, siempre que haya más películas en esta lista. Se realiza llamando a la petición de la *API* para obtener una colección concreta de forma paginada, es decir por medio de páginas.

Para su desarrollo hemos utilizado un *adapter*, como los mencionados anteriormente, en concreto un *GridView*, que nos permite especificar el número de ítems o elementos que aparecerán por cada fila.

5.4.9 Búsqueda de películas

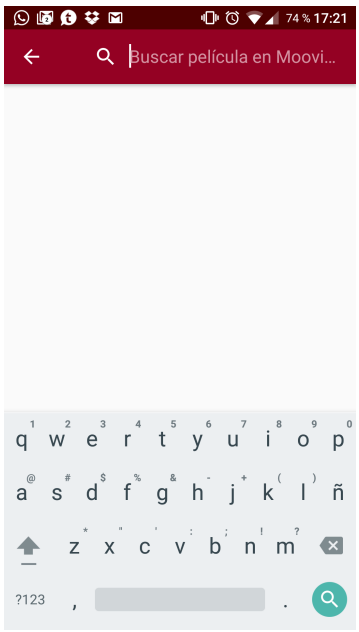


Figura 5.22 Página para la búsqueda de películas

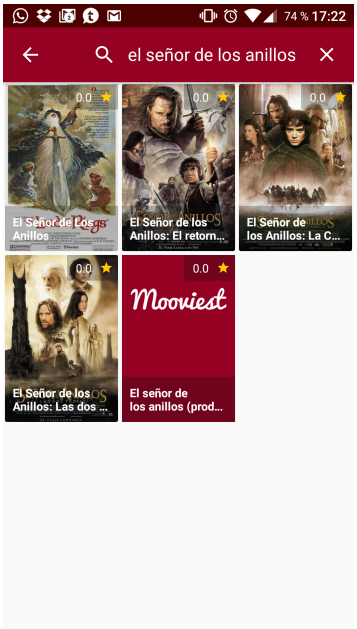


Figura 5.23 Resultado de una búsqueda

En esta vista correspondiente a la ***Figura 5.22***, tenemos la opción de buscar una película por su título original o en el idioma que tenga el usuario. Al igual que en las listas de un usuario se cargan máximo diez películas; si hubiera más resultados y se hace *scroll* hacia abajo, se volvería a realizar una petición a la *API* por la siguiente página.

Una vez tenemos el listado también podemos seleccionar una y ser redirigidos a su vista de detalle.

Para su desarrollo se ha utilizado el mismo sistema que para una lista completa de películas del usuario.

5.4.10 Detalle de una película

Figura 5.24 Vista de detalle de una película

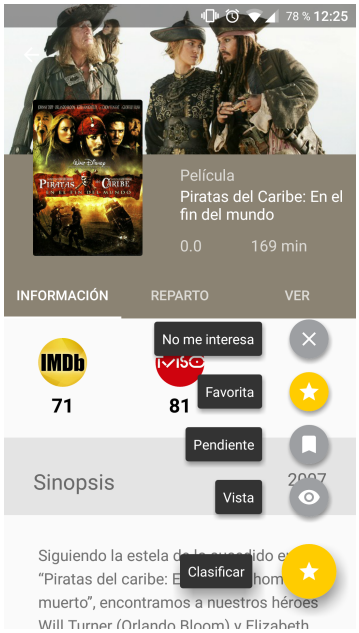


Figura 5.25 Botones de clasificación en la vista de detalle

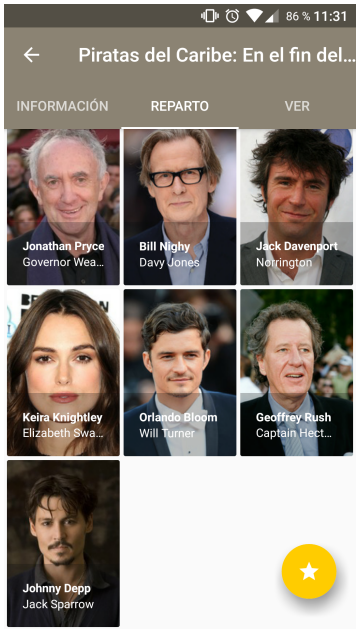


Figura 5.26 Reparto de la película

Esta vista corresponde a la *activity* ***MovieDetailActivity.java***, y en ella mostramos la información detallada de una película como su carátula, sinopsis, título, duración, géneros, reparto, etc. Además, mostramos las puntuaciones de varias plataformas como son *IMDb* y *Tviso* y la opción de clasificar la película en una lista o cambiarla si ya la teníamos clasificada. Todo ello lo podemos ver en la ***Figura 5.24***, la ***Figura 5.25*** y la ***Figura 5.26***.

Esto último lo hacemos mediante un botón flotante en Android ***FloatingButton***. Como hemos comentado aparecería un icono de '+', si la película aún no está clasificada, o el icono correspondiente a la lista en la lo está. Al pulsarlo nos aparecería un menú con las posibles opciones y, al pulsar en una que no sea en la que está actualmente, cambiaría de lista.

Sobre el desarrollo al clasificar una película llamaríamos a las funciones de la API para añadirla a una colección o actualizar la que tenía, ***PostMovieCollection*** o ***PatchMovieCollection*** respectivamente.

La vista si divide en *tabs* como en la pantalla de *Home* (**punto 5.4.2**) y se ha desarrollado de la misma forma con ***ViewPager***. El reparto se ha realizado de la misma forma que una lista de películas, es decir, mediante un *adapter*, pero esta vez mostrando la foto del personaje, si nombre real y su nombre de reparto.

De la misma forma, las puntuaciones de las plataformas se han realizado como las listas de películas del *Home* con *scroll* horizontal.

5.4.11 Cliente REST

En este apartado se va a explicar cómo se ha creado el cliente *REST*, encargado de realizar las peticiones a la *API*. Hemos utilizado una librería externa llamada *Retrofit*, que nos permite realizar este cometido; en concreto, *Retrofit*, adapta una interfaz *Java*, definiendo los *endpoints* mediante métodos y uso de anotaciones, para las llamadas *HTTP* a la *API*.

Un ejemplo de un método o endpoint sería el siguiente, en el que se realiza una petición *GET* para obtener el perfil de un usuario mediante su *ID*.

```
public interface MooviestApiInterface {

   @GET("users/{id}/")
   Call<UserProfileResponse> getUserProfile(@Path("id") int id);
                  .
                  .
                  .
}
```

Para la utilización de esta interfaz, debemos crear un objeto *Retrofit Builder,* como vemos en el siguiente fragmento de código:

```
String baseAPIUrl = "http://localhost:8000/api/";

public static final Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(baseAPIUrl)
        .addConverterFactory(GsonConverterFactory.create())
        .client(okHttpClient)
        .build();
```

A este objeto le pasamos la *URL* base de la *API*, además del tipo de conversión para obtener las respuestas a las peticiones, que en este caso usamos *Gson*, una librería para convertir objetos *Java* en *JSON* y viceversa. Además, le pasamos un cliente *HTTP*, que creamos con la librería *OkHttp*, pasándole las cabeceras y la autorización por *token* en el caso de que sea necesario para la llamada a la *API*.

5.4.12 Llamadas a la API

Para la realización de una llamada a la API haremos uso del cliente creado en el punto anterior. Veremos un ejemplo una llamada a la API, para el mismo caso de obtener el perfil de un usuario.

```
MooviestApiInterface apiInterface =
MooviestApiInterface.retrofit.create(MooviestApiInterface.class);

Call<UserProfileResponse> call =
apiInterface.getUserProfile(userId);

UserProfileResponse result = call.execute().body();
```

Creamos la implementación de la interfaz *API*, instanciamos un objeto *Call* con el método concreto de la interfaz y lo ejecutamos para que nos devuelva el resultado de la petición.

Todo este proceso lo realizamos en un *AsyncTask*, es decir, una hebra distinta a la del hilo principal, para que no se bloquee nuestra actividad principal donde se encuentra el usuario.

Capítulo 6. Conclusiones y trabajo futuro

6.1 Conclusiones

En este trabajo de fin de grado se han descrito todas las fases para poder desarrollar una aplicación multiplataforma funcional desde cero, con tecnologías de servidor, como *Django*, y móvil, *Android*.

Hoy en día, podemos ver el gran uso de los dispositivos móviles como herramienta de trabajo diario, y lo fundamental para cualquier usuario, es poder tener toda la información en la palma de su mano, esté donde esté. En esta *app*, permitimos al usuario acceder y mantener una biblioteca y lista de tus películas clasificadas, así como acceder a toda la información asociada a ella.

Se ha explicado en detalle cómo se iba a solucionar este problema, al tener toda la información distribuida entre multitud de plataformas y muy fragmentada en Internet. Se han detallado todas las fases de análisis y recopilación de información y de la estructura de la aplicación. Una vez definido esto, pudimos comenzar todo el desarrollo necesario para llevarlo a cabo.

Haciendo uso de tecnologías de servidor como *Django* y *Django REST Framework* para crear la *API*, pudimos construir, junto con una base datos en *PostgreSQL*, una arquitectura de servidor bastante completa, para dar respuesta a nuestros clientes en *Android* e *iOS*.

Para conseguir rendimientos superiores y mayor escalabilidad en un futuro, pensamos que la mejor opción era desarrollar nuestras aplicaciones en sus plataformas nativas, en mi caso en Android. De esta forma no depender de otras tecnologías web emergentes, como el caso de *Ionic*, que, aunque te faciliten el desarrollo inicial, al poder desarrollar en varias plataformas, puede llegar a ser tedioso adaptar tus *apps* a nuevas actualizaciones de los sistemas y plataformas.

La realización de este proyecto, ha supuesto un aumento de mis conocimientos en muchas áreas del desarrollo software, ya que hemos desarrollado aplicaciones de servidor, una *API*, el desarrollo de *scripts* y *scrappers* para la recogida de información de *APIs* y páginas webs, una base de datos y una aplicación móvil nativa. Sobre todo, hemos aprendido a afrontar un problema real y su solución, usando una metodología ágil, tecnologías colaborativas para trabajar en grupo, gestionar un software mediante control de versiones y muchos otros aspectos del desarrollo software.

En resumen, he disfrutado al formar parte de este proyecto en equipo y al desarrollar una aplicación totalmente funcional y usable desde su primera versión. Consideramos también que el resultado es una aplicación útil para el público aficionado al cine, que necesitan clasificar su contenido, para posteriormente tenerlo accesible rápidamente.

6.2 Trabajo futuro

- **Sistema de recomendaciones:** Para atraer a un mayor público en el uso de nuestra aplicación, se había pensado en el desarrollo de un sistema de recomendaciones, basado en técnicas de Aprendizaje Automático (*Machine Learning*). Es decir, nuestra app aprenderá del usuario y del contenido que ya hemos clasificado, para así recomendar nuevas películas y contenidos. De esta forma, cuanto más se use la aplicación y se clasifiquen las películas, nuestro sistema será más fiable en sus recomendaciones.

- **Iniciar sesión con redes sociales:** Muchos usuarios ven bastante tedioso registrarte en una web o aplicación por medio de un formulario, y muchos de ellos, no llegan a utilizarlas por este inconveniente. Las redes sociales nos facilitan esta tarea, obteniendo nuestra información de ellas y así, no tener que volver a introducirla en nuevas aplicaciones.

- **Recuperar contraseña:** Una buena manera de hacer que los usuarios confíen en nuestra plataforma, es poder implementar esta función para poder recuperar su contraseña, si ésta se ha olvidado.

- **Confirmar cuenta por *email*:** Desarrollaremos este paso para validar que los usuarios que se registren, sean usuarios reales de nuestra aplicación y evitar que *bots* o máquinas creen usuarios falsos.

- **Consultar las películas en las que ha participado una *celebrity*:** Además de nuestro sistema de recomendaciones, sería de gran utilidad, consultar las películas en las que ha participado tu actor o director favorito. De esta forma tendríamos una funcionalidad extra, para que el usuario vea su contenido favorito.

- **Buscar y seguir usuarios y *celebrities*:** Además de buscar películas y clasificarlas, añadiremos la función de buscar usuarios y *celebrities*, y seguirlos. Esta información aparecerá en nuestro perfil con los apartados: siguiendo, seguidores y *celebrities*.

- **Consultar el perfil y las listas de usuarios:** También añadiremos la funcionalidad de consultar las listas de películas clasificadas de un usuario.

Bibliografía

[1] Alliance, G. a. (2012-2016). *Android Developers*. Obtenido de Developer Android: https://developer.android.com/index.html

[2] Bahit, E. (2012). *Curso: Python para principiantes.* Buenos Aires, Argentina: Safe Creative.

[3] Christie, T. (de de 2011-2016). *Django REST Framework*. Obtenido de Django REST Framework: http://www.django-rest-framework.org/

[4] Consumerlab, T. M. (2014). *Ericsson.* Obtenido de Ericsson: https://www.ericsson.com/res/docs/2014/consumerlab/tv-media-2014-ericsson-consumerlab.pdf

[5] Foundation, D. S. (2005-2016). *The Web framework for perfectionists with deadlines | Django*. Obtenido de Django Project: https://www.djangoproject.com/

[6] GitHub, I. (2008-2016). *GitHub*. Obtenido de GitHub: https://github.com/

[7] Google. (2014-2016). *Introduction - Material design - Material design guidelines*. Obtenido de Material Google: https://material.google.com/

[8] Lequerica, J. R. (2016). *Desarrollo De Aplicaciones Para Android - Edición 2017 (Manuales Imprescindibles).* ANAYA.

[9] PosgreSQL, R. M. (2009-2016). *postgresql.org.es*. Obtenido de PostgreSQL: http://www.postgresql.org.es/

[10] Richardson, C. a. (de de 2004-2016). *Beautiful Soup: We called him Tortoise because he taught us.* Obtenido de Beautiful Soup: https://www.crummy.com/software/BeautifulSoup/

[11] Square. (2013-2016). *Picasso*. Obtenido de Picasso: http://square.github.io/picasso/

[12] Square. (2013-2016). *Retrofit*. Obtenido de Retrofit: https://square.github.io/retrofit/

[13] W3C. (1996-2016). *World Wide Web Consortium (W3C)*. Obtenido de W3: https://www.w3.org/